ΤΕΧΝΟΛΟΓΙΚΟ ΕΚΠΑΙΔΕΥΤΙΚΟ ΙΔΡΥΜΑ ΚΡΗΤΗΣ

ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΦΑΡΜΟΓΩΝ
ΤΜΗΜΑ ΕΦΑΡΜΟΣΜΕΝΗΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΠΟΛΥΜΕΣΩΝ

ΔΥΝΑΜΙΚΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΚΑΙ ΕΦΑΡΜΟΓΕΣ

Δατσέρης Γιάννης
ΑΜ: 1280

Επιβλέπων καθηγητής
Τριανταφυλλίδης Γεώργιος

Περιεχόμενα

Κεφάλαιο 1ο
(Εισαγωγή)

Εισαγωγή

Αν η λύση ενός προβλήματος μπορεί να εκφραστεί μαθηματικά με αναδρομικό τρόπο, τότε το πρόβλημα μπορεί να λυθεί από ένα αναδρομικό αλγόριθμο. Συχνά οι μεταγλωττιστές γλωσσών προγραμματισμού συντείνουν ώστε η εκτέλεση πολλών αναδρομικών προγραμμάτων να μην είναι αποδοτική. Σε τέτοιες περιπτώσεις μπορούμε να 'βοηθήσουμε' το μεταγλωττιστή μετατρέποντας τον αλγόριθμο σε μη-αναδρομικό αλγόριθμο ο οποίος συστηματικά φυλάει απαντήσεις υποπροβλημάτων σε ένα πίνακα. Μια τεχνική η οποία χρησιμοποιεί αυτή τη μέθοδο είναι γνωστή ως *δυναμικός προγραμματισμός (dynamic programming)*.

1.1 Ιστορία

Ο όρος χρησιμοποιήθηκε αρχικά στη δεκαετία του '40 από τον Richard Bellman για να περιγράψει τη διαδικασία επίλυσης προβλημάτων. Μέχρι το 1953, το είχε καθορίσει με την σύγχρονη έννοια. Η συμβολή του Bellman αναφέρεται στο όνομα της Bellman εξίσωσης, ένα κεντρικό αποτέλεσμα του δυναμικού προγραμματισμού που επαναδιατυπώνει ένα πρόβλημα βελτιστοποίησης με επαναλαμβανόμενη μορφή.

Αρχικά η λέξη «προγραμματισμός» στο «δυναμικό προγραμματισμό» δεν είχε καμία σύνδεση με τον προγραμματισμό υπολογιστών, και προήλθε αντ' αυτού από τον όρο «μαθηματικός προγραμματισμός» - ένα συνώνυμο για τη βελτιστοποίηση. Εντούτοις, σήμερα πολλά προβλήματα βελτιστοποίησης λύνονται καλύτερα με την δημιουργία ενός προγράμματος στον υπολογιστή, που εφαρμόζει έναν δυναμικό αλγόριθμο προγραμματισμού, παρά την πραγματοποίηση των εκατοντάδων κουραστικών υπολογισμών με το χέρι. Ο δυναμικός προγραμματισμός είναι μια σχεδιαστική περισσότερο τεχνική ,χρήσιμη στην επίλυση πολύπλοκων προβλημάτων.

1.2 Περιγραφή προβλημάτων

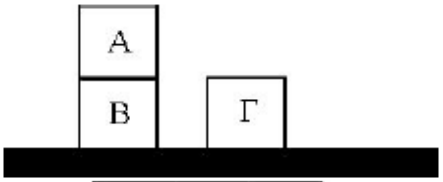
Η έννοια του προβλήματος είναι διαισθητικά γνωστή σε όλους: υπάρχει μια δεδομένη κατάσταση (αρχική), υπάρχει μια επιθυμητή κατάσταση (τελική) και διαθέσιμες ενέργειες που πρέπει να γίνουν ώστε να προκύψει η επιθυμητή. Στην καθημερινότητα χρησιμοποιείται με λάθος τρόπο η λέξη «πρόβλημα», ακόμη και όταν δεν είναι γνωστό ένα από τα παραπάνω τρία βασικά συστατικά. Αν ,λόγου χάρη, κάποιος δεν έχει στόχους ή δεν γνωρίζει ποιες ενέργειες μπορεί να εκτελέσει, τότε δεν

υφίσταται πρόβλημα, τουλάχιστον με την τυπική έννοια του όρου. Όμως, η επίλυση προβλημάτων που επιδιώκεται από την Τεχνητή Νοημοσύνη απαιτεί τον τυποποιημένο και σαφή ορισμό τους. Επιπλέον, ο ορισμός ενός προβλήματος πρέπει να είναι ανεξάρτητος από την πολυπλοκότητα επίλυσης του, ενώ η πολυπλοκότητα του καθορίζεται από την πολυπλοκότητα του αλγορίθμου αναζήτησης που εφαρμόζεται για την επίλυση του.

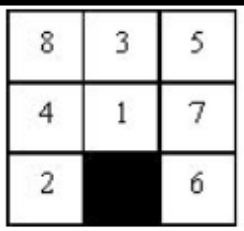
Ορισμένα προβλήματα, για παράδειγμα, είναι οι κύβοι (blocks), το N-puzzle, η τρίλιζα (tic-tac-toe), ο λαβύρινθος (maze), οι πύργοι του Ανόι (Hanoi towers), οι κανίβαλοι και οι ιεραπόστολοι (missionaries and cannibals) και τα ποτήρια (water glass). Πρόκειται δηλαδή για απλά προβλήματα, των οποίων η παρουσίαση βοηθάει στην κατανόηση των αρχών επίλυσης προβλημάτων. Άλλα παραδείγματα πιο πολύπλοκων προβλημάτων είναι το σκάκι (chess), ο πλανόδιος πωλητής (traveling salesperson), οι N-βασίλισσες (N-queens).

ΕΙΚΟΝΑ 1.1 ΠΑΡΑΔΕΙΓΜΑΤΑ ΠΡΟΒΛΗΜΑΤΩΝ

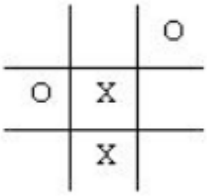
κύβοι (blocks)



N-puzzle

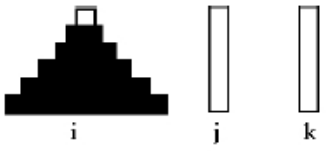


τρίλιζα (tic-tac-toe)

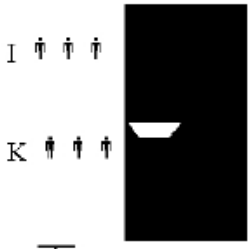


λαβύρινθος (maze)

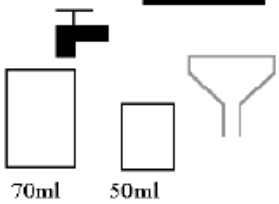
πύργοι του Ανόι (Hanoi towers)



κανίβαλοι και ιεραπόστολοι (missionaries and cannibals)



ποτήρια (water glass)



1.3 Αλγόριθμοι αναζήτησης

Δοθέντος ενός προβλήματος με περιγραφή στο χώρο καταστάσεων ή με την αναγωγή, στόχος είναι να βρεθεί η λύση του. Η τυποποίηση της περιγραφής ενός προβλήματος διευκολύνει την επίλυση του. Κατ' αντιστοιχία, η τυποποίηση των βημάτων επίλυσης διευκολύνει την αυτοματοποίηση, δηλαδή την υλοποίηση του τρόπου επίλυσης σε ένα υπολογιστικό σύστημα. Η αυτοματοποίηση αυτή επιτυγχάνεται μέσω αυστηρά προκαθορισμένων βημάτων, δηλαδή αλγορίθμων, που πρέπει να εφαρμοστούν για να επιλυθεί ένα πρόβλημα. Επειδή οι αλγόριθμοι αυτοί αναζητούν τη λύση στο πρόβλημα ονομάζονται ***αλγόριθμοι αναζήτησης (search algorithms)***. Υπάρχουν αρκετοί αλγόριθμοι και πολλές φορές συγκεκριμένα προβλήματα δίνουν την αφορμή για τον σχεδιασμό νέων. Ωστόσο, οι αλγόριθμοι που παρουσιάζονται είναι οι γνωστότεροι και αποτελούν βασικά δομικά στοιχεία που οδηγούν στη δημιουργία νέων αλγορίθμων. Στον παρακάτω πίνακα παρουσιάζονται γνωστοί αλγόριθμοι μερικοί εκ των οποίων θα αναλυθούν παρακάτω.

ΠΙΝΑΚΑΣ 1 ΑΛΓΟΡΙΘΜΟΙ

Όνομα Αλγορίθμου	**Συντομογραφία**	**Ελληνική Ορολογία**
Deep – First Search	DFS	Αναζήτηση Πρώτα σε Βάθος
Breadth – First Search	BFS	Αναζήτηση Πρώτα σε Πλάτος
Iterative Deepening	ID	Επαναληπτική Εκβάθυνση
Bi-directional Search	BiS	Αναζήτηση Διπλής Κατεύθυνσης

Branch and Bound	B&B	Επέκταση και Οριοθέτηση
Hill Climbing	HC	Αναρρίχηση Λόφου
Enforced Hill Climbing	EHC	Εξαναγκασμένη Αναρρίχηση Λόφων
Simulated Annealing	SA	Προσομοιωμένη Ανόπτηση
Tabu Search	TS	Αναζήτηση με Απαγορευμένες Καταστάσεις
Beam Search	BS	Ακτινωτή Αναζήτηση
Best – First Search	BestFS	Αναζήτηση Πρώτα στο Καλύτερο
A* (A star)	A*	A* (Αλφα Αστρο)
Iterative Deepening A*	IDA*	A* με Επαναληπτική Εκβάθυνση
Minmax	Minmax	Αναζήτηση Μέγιστου-Ελάχιστου
Alpha-Beta	AB	Άλφα-Βήτα

1.4 Περιγραφή στο χώρο καταστάσεων και στο χώρο αναζήτησης

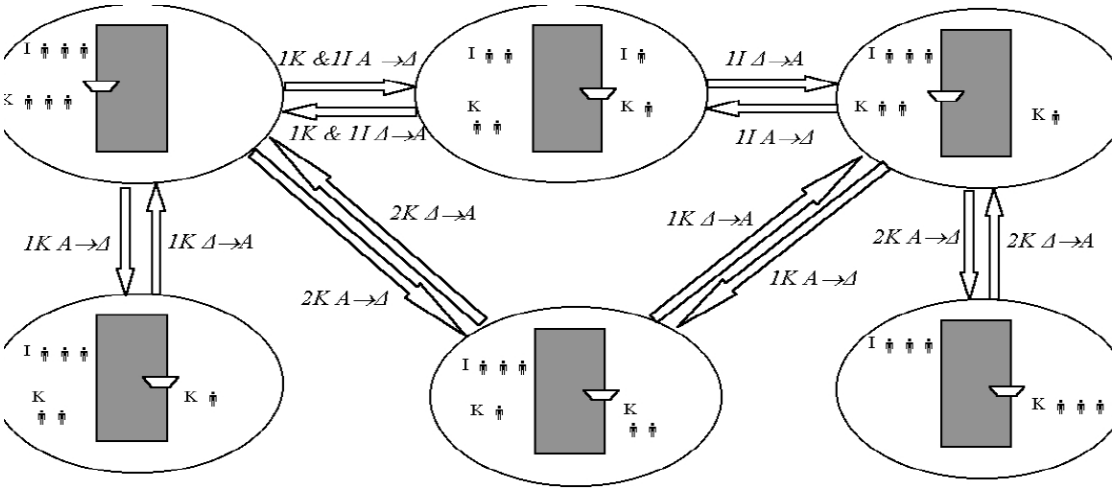
Ο *κόσμος ενός προβλήματος* (problem world) αποτελείται μόνο από τα αντικείμενα που υπάρχουν σε αυτόν, τις ιδιότητες των αντικειμένων και τις σχέσεις που τα συνδέουν. Άρα ο κόσμος ενός προβλήματος είναι υποσύνολο του πραγματικού κόσμου και περιέχει μόνον όσα αντικείμενα έχουν άμεση σχέση με το πρόβλημα. Η κατάσταση ενός κόσμου είναι ένα *στιγμιότυπο* (instance) ή *φωτογραφία* (snapshot) μιας συγκεκριμένης χρονικής στιγμής κατά την εξέλιξη του κόσμου. Παρατηρώντας μια κατάσταση είναι φανερό τι ακριβώς συμβαίνει στον κόσμο.

Οι καταστάσεις ενός κόσμου συνδέονται μεταξύ τους, με την έννοια ότι από μια κατάσταση μπορεί να προκύψει μια νέα κατάσταση. Η παραγωγή νέων καταστάσεων οφείλεται στους *τελεστές μετάβασης* ή *ενέργειες* που μπορεί να εφαρμοστούν σε κάποια κατάσταση του κόσμου του προβλήματος. *Τελεστής μετάβασης* (transition operator) είναι μια αντιστοίχηση μιας κατάστασης του κόσμου σε νέες καταστάσεις.

Χώρος καταστάσεων (state space ή domain space) ενός προβλήματος ονομάζεται το σύνολο όλων των έγκυρων καταστάσεων. Ο χώρος καταστάσεων απεικονίζεται συνήθως διαισθητικά με ένα *γράφο (graph)*. Για παράδειγμα, το παρακάτω σχήμα (Σχήμα 1.1) δείχνει μέρος του χώρου καταστάσεων για τον κόσμο των ιεραποστόλων και των

κανιβάλων. Κάθε κόμβος είναι μια κατάσταση και κάθε ακμή που ενώνει δυο κόμβους είναι ένας τελεστής. Τυπικά, ο χώρος καταστάσεων μπορεί να αναπαρασταθεί με τη περιγραφή σε μια κατάλληλη γλώσσα, όπως για παράδειγμα τη λογική.

ΣΧΗΜΑ 1.1 ΛΥΣΗ ΠΡΟΒΛΗΜΑΤΟΣ ΙΕΡΑΠΟΣΤΟΛΩΝ-ΚΑΝΙΒΑΛΩΝ



Ένα πρόβλημα ορίστηκε προηγουμένως ως η τετράδα (I,G,T,S) ,όπου το S είναι ο χώρος καταστάσεων, I είναι η αρχική κατάσταση, G είναι το σύνολο των τελικών καταστάσεων και T είναι το σύνολο των τελεστών μετάβασης. Σκοπός ενός αλγορίθμου αναζήτησης είναι να προσπαθήσει να βρει τουλάχιστον μια λύση μέσα στον χώρο καταστάσεων. Φυσιολογικά ένας τέτοιος αλγόριθμος πρέπει να εξετάσει μόνον το υποσύνολο του χώρου καταστάσεων το οποίο αφορά στην αρχική κατάσταση. Δοθέντος ενός προβλήματος (I,G,T,S), χώρος αναζήτησης (search space) SP είναι το σύνολο όλων των καταστάσεων που είναι προσβάσιμες από την αρχική κατάσταση. Τυπικά, μια κατάσταση s ονομάζεται προσβάσιμη (accessible) αν υπάρχει μια ακολουθία τελεστών μετάβασης $t_1,t_2,...,t_k$,που ανήκουν στο T, τέτοια ώστε $s=t_k(\ldots(t_2(t_1(I)))\ldots)$.

Η διαφορά μεταξύ του χώρου καταστάσεων και του χώρου αναζήτησης είναι λεπτή. Ο χώρος αναζήτησης είναι υποσύνολο του χώρου καταστάσεων. Η διαφορά έγκειται στο γεγονός ότι εξ' ορισμού ο χώρος αναζήτησης εξαρτάται από την αρχική κατάσταση, ενώ ο χώρος καταστάσεων όχι. Μόνο όταν όλες οι καταστάσεις του χώρου καταστάσεων είναι προσβάσιμες από την αρχική κατάσταση δυο αυτοί χώροι ταυτίζονται.

1.5 Διαδικασία επιλογής ενός αλγόριθμου αναζήτησης

Δοθέντος ενός προβλήματος είναι σημαντικό να επιλεγεί ο καταλληλότερος αλγόριθμος για την επίλυση του. Η επιλογή αυτή γίνεται βάσει κάποιων κριτηρίων, τα οποία όμως δεν μπορεί να τυποποιηθούν. Η επιλογή εξαρτάται κυρίως από την φύση του προβλήματος και σε μεγάλο βαθμό από τους συμβιβασμούς που πρέπει να γίνουν. Για παράδειγμα, μπορεί κάποιος να είναι διατεθειμένος να θυσιάσει την αποδοτικότητα σε χώρο ή χρόνο προς χάριν της καλύτερης λύσης ή να θυσιάσει την πληρότητα, επιδιώκοντας την γρήγορη εύρεση οποιασδήποτε λύσης. Εν συντομία, η επιλογή ενός αλγορίθμου βασίζεται στα εξής κριτήρια:

- Τον αριθμό των καταστάσεων που αυτός επισκέπτεται
- Την δυνατότητα εύρεσης λύσεων εφόσον αυτές υπάρχουν
- Τον αριθμό των λύσεων
- Την ποιότητα των λύσεων
- Την αποδοτικότητα σε χρόνο
- Την αποδοτικότητα σε χώρο (στην μνήμη του συστήματος)
- Την ευκολία υλοποίησης του

Στα κριτήρια αυτά εντάσσεται και η έννοια του ***κλαδέματος*** ή ***αποκοπής καταστάσεων (pruning)*** του χώρου αναζήτησης. Αποκοπή είναι η διαδικασία κατά την οποία ο αλγόριθμος απορρίπτει, κάτω από ορισμένες συνθήκες, κάποιες καταστάσεις και μαζί με αυτές όλες τις επόμενες καταστάσεις που εξαρτώνται από αυτήν. Η αποκοπή μπορεί να βασίζεται είτε σε αντικειμενικά κριτήρια, όταν είναι σίγουρο ότι δεν υπάρχει λύση από εκεί και κάτω ή σε αυθαίρετα κριτήρια. Για παράδειγμα, αν και η συνέχιση της αναζήτησης από μια κατάσταση μπορεί να οδηγήσει σε λύση, το κόστος υπολογισμού της ίσως να είναι υπερβολικά μεγάλο, με αποτέλεσμα να αποφασιστεί να κλαδευτεί ο χώρος αναζήτησης που συνδέεσαι με αυτήν την κατάσταση.

1.6 Αλγόριθμοι τυφλής αναζήτησης

Οι αλγόριθμοι τυφλής αναζήτησης (blind search algorithm) εφαρμόζονται σε προβλήματα στα οποία δεν υπάρχει πληροφορία που να επιτρέπει την αξιολόγηση των καταστάσεων του χώρου αναζήτησης. Έτσι οι αλγόριθμοι αυτοί αντιμετωπίζουν με τον ίδιο ακριβώς τρόπο οποιοσδήποτε πρόβλημα καλούνται να λύσουν. Για τους αλγορίθμους τυφλής αναζήτησης, το τι απεικονίζει κάθε κατάσταση του προβλήματος

είναι παντελώς αδιάφορο. Σημασία έχει η χρονική σειρά με την οποία παράγονται οι καταστάσεις από το μηχανισμό επέκτασης. Δύο γενικές μέθοδοι επίλυσης, οι οποίες εξετάζουν τον χώρο με συστηματικό ,τυφλό τρόπο είναι η *αναζήτηση πρώτα σε βάθος* (*Depth First Search - DFS*) και η *αναζήτηση πρώτα σε πλάτος* (*Breadth First Search - BFS*)

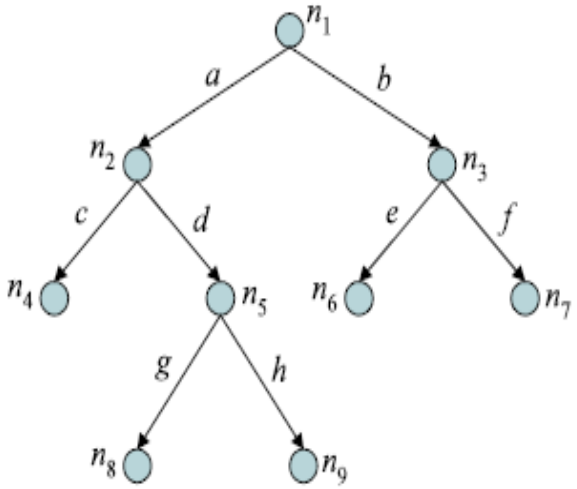
1.6.1 Αναζήτηση πρώτα σε πλάτος

Ίσως η πιο απλή μέθοδος αναζήτησης χωρίς πληροφορία είναι η *αναζήτηση κατά πλάτος* (*breadth–first search*). Σύμφωνα με αυτήν την μέθοδο, πρώτα επεκτείνεται η ρίζα με την εφαρμογή όλων των τελεστών, κατόπιν επεκτείνονται όλοι οι άμεσοι διάδοχοι της ρίζας, κατόπιν όλοι οι διάδοχοι αυτών και η διαδικασία συνεχίζεται μέχρι να επεκταθούν όλοι οι κόμβοι. Γενικά, όλοι οι κόμβοι βάθους *d* επεκτείνονται πριν από τους κόμβους βάθους *d*+1.

Αναλυτικά η περιγραφή του αλγορίθμου είναι:

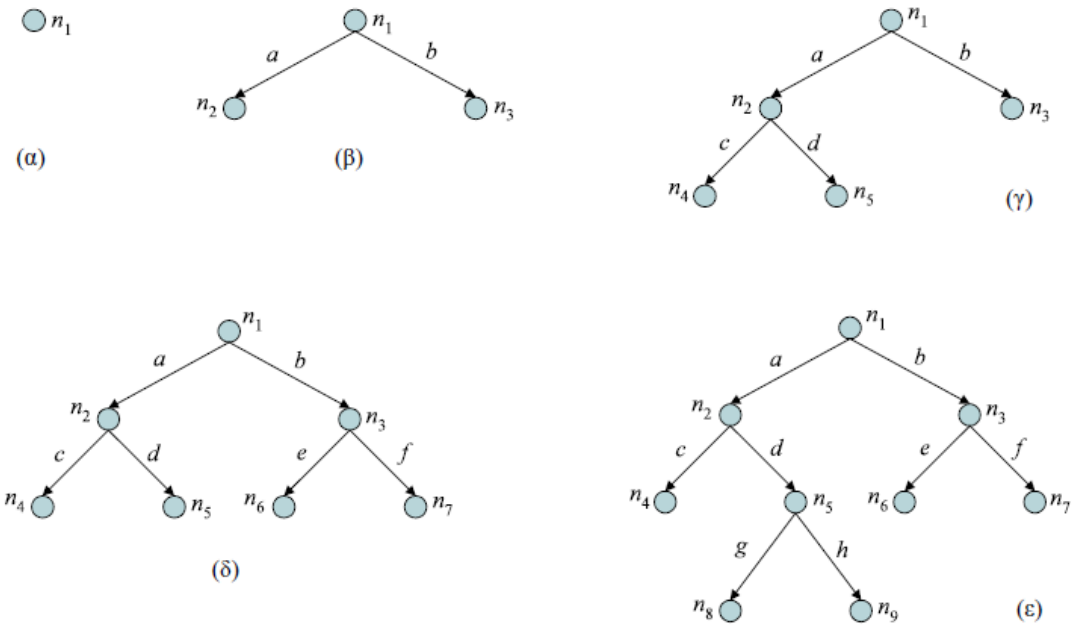
1. Βάλε την αρχική κατάσταση στο μέτωπο της αναζήτησης.
2. Αν το μέτωπο της αναζήτησης είναι κενό τότε σταμάτησε.
3. Βγάλε την πρώτη κατάσταση από το μέτωπο της αναζήτησης.
4. Αν είναι η κατάσταση μέλος του κλειστού συνόλου τότε πήγαινε στο βήμα 2.
5. Αν η κατάσταση είναι μια τελική τότε ανέφερε την λύση.
6. Αν θέλεις και άλλες λύσεις πήγαινε στο βήμα 2. Αλλιώς σταμάτησε.
7. Εφάρμοσε τους τελεστές μετάβασης για να βρεις τις καταστάσεις-παιδιά.
8. Βάλε τις καταστάσεις-παιδιά στο τέλος του μετώπου της αναζήτησης.
9. Βάλε την κατάσταση-γονέα στο κλειστό σύνολο.
10. Πήγαινε στο βήμα 2.

ΣΧΗΜΑ 1.2 ΔΕΝΤΡΟ



Στο παρακάτω σχήμα (Σχήμα 1.3), φαίνεται η πρόοδος της μεθόδου στο δέντρο που χρησιμοποιήθηκε στο διπλανό δέντρο (Σχήμα 1.2). Η συγκεκριμένη μέθοδος είναι πολύ συστηματική, αφού εξετάζει εξαντλητικά όλα τα μονοπάτια του ίδιου ύψους πριν προχωρήσει στο επόμενο, δηλαδή πρώτα εξετάζονται όλα τα μονοπάτια ύψους 1, έπειτα όλα τα μονοπάτια ύψους 2, κτλ.

ΣΧΗΜΑ 1.3 ΑΝΑΠΤΥΞΗ ΜΕΘΟΔΟΥ BFS



Βασικό πλεονέκτημα του BFS είναι ότι βρίσκει πάντα τη μικρότερη σε μήκος λύση, δηλαδή αυτή με τους λιγότερους τελεστές. Η

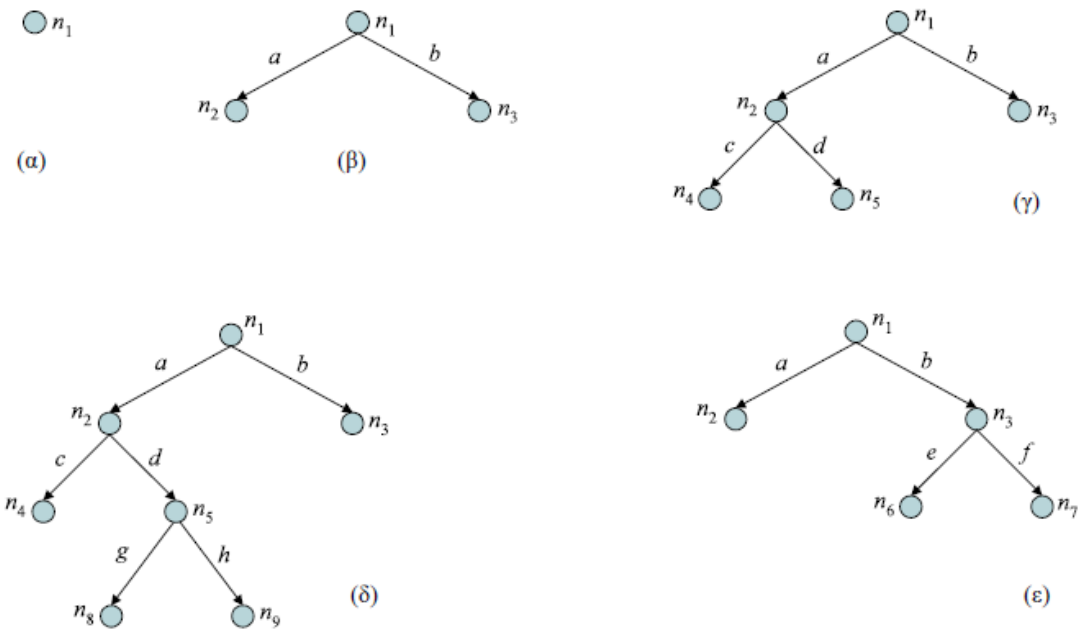
λύση αυτή μπορεί να θεωρηθεί βέλτιστη, μόνο εάν όλοι οι τελεστές έχουν το ίδιο κόστος. Επιπλέον, ο BFS είναι πλήρης, δηλαδή θα βρει λύση σε κάποιο πρόβλημα , σε περίπτωση που υπάρχει λύση. Σε μια ακραία περίπτωση, όταν ένα δέντρο αναζήτησης έχει άπειρο πλάτος, δηλαδή υπάρχουν άπειροι τελεστές που εφαρμόζονται σε μια κατάσταση, τότε ο BFS έχει πρόβλημα στην ανάπτυξη του δέντρου αναζήτησης.

Βασικό μειονέκτημα του BFS είναι ότι το μέτωπο της αναζήτησης μεγαλώνει πολύ σε μέγεθος, οπότε και σε απαιτήσεις μνήμης για την αποθήκευση των προς επέκταση καταστάσεων, οι οποίες μεγαλώνουν εκθετικά με το βάθος αναζήτησης. Ο BFS, ακόμη και χωρίς έλεγχο βρόχων, θα έβρισκε την καλύτερη λύση στο πρόβλημα, αλλά το μέτωπο αναζήτησης θα αύξανε πάρα πολύ. Γενικά δεν μπορεί κανείς να εκφράσει προτίμηση για τον BFS έναντι του DFS παρά μόνον εάν ξέρει τη μορφή του χώρου αναζήτησης.

1.6.2 Αναζήτηση πρώτα σε βάθος

Η *αναζήτηση κατά βάθος* (*depth–first search DFS*) επεκτείνει έναν κόμβο κάθε φορά μέχρι το βαθύτερο επίπεδο του δένδρου, αφήνοντας ταυτόχρονα ένα ίχνος σε κάθε κόμβο που δεν έχει ακόμα επεκταθεί. Όταν η αναζήτηση φτάσει σε φύλλο, τότε γυρίζει πίσω στο κοντινότερο ίχνος και η διαδικασία επαναλαμβάνεται μέχρι να βρεθεί μία λύση (Σχήμα 1.4). Διαγραμματικά για να κατανοηθεί καλυτέρα η διαφορά ανάμεσα στους δυο αλγορίθμους ,δηλαδή τον BFS και τον DFS θα χρησιμοποιηθεί το δέντρο του Σχήματος 1.2.

ΣΧΗΜΑ 1.4 ΑΝΑΠΤΥΞΗ ΜΕΘΟΔΟΥ DFS



Αναλυτικά η περιγραφή του αλγορίθμου είναι:

1. Βάλε την αρχική κατάσταση στο μέτωπο της αναζήτησης.
2. Αν το μέτωπο της αναζήτησης είναι κενό τότε σταμάτησε.
3. Βγάλε την πρώτη κατάσταση από το μέτωπο της αναζήτησης.
4. Αν είναι η κατάσταση μέλος του κλειστού συνόλου τότε πήγαινε στο βήμα 2.
5. Αν η κατάσταση είναι μια τελική τότε ανέφερε την λύση.
6. Αν θέλεις και άλλες λύσεις πήγαινε στο βήμα 2. Αλλιώς σταμάτησε.
7. Εφάρμοσε τους τελεστές μετάβασης για να βρεις τις καταστάσεις-παιδιά.
8. Βάλε τις καταστάσεις-παιδιά στην αρχή του μετώπου της αναζήτησης.
9. Βάλε την κατάσταση-γονέα στο κλειστό σύνολο.
10. Πήγαινε στο βήμα 2.

Η μέθοδος έχει πολύ μικρές απαιτήσεις σε μνήμη, σε αντίθεση με την BFS, αφού χρειάζεται μόνο να διατηρείται το τρέχον μονοπάτι και τα διάφορα ίχνη. Η DFS είναι γενικά πιο γρήγορη από την BFS, διότι στην περίπτωση πολλών λύσεων υπάρχει μεγάλη πιθανότητα να βρεθεί μία από αυτές με αναζήτηση μόνο σε ένα μικρό κομμάτι του χώρου αναζήτησης.

Το μέτωπο αναζήτησης είναι μια δομή στοίβας (Stack LIFO, Last In First Out) ,δηλαδή οι νεές καταστάσεις τοποθετούνται πάντα στην κορυφή της στοίβας και η αναζήτηση συνεχίζεται με μια από αυτές.

Ένα από τα βασικά πλεοντεκτήματα του DFS είναι ότι έχει μικρές απαιτήσεις σε χώρο. Το μέτωπο της αναζήτησης δεν μεγαλώνει πάρα πολύ και έτσι η μνήμη που χρειάζεται για να θυμάται τις καταστάσεις προς επέκταση είναι σχετικά μικρή.

Το βασικό μειονέκτημα του DFS είναι ότι δεν εγγυάται ότι η πρώτη λύση που θα βρεθεί θα είναι η βέλτιστη (μονοπάτι με το μικρότερο μήκος ή κόστος). Επίσης, αν δεν υπάρχει έλεγχος βρ΄χων ή αν ο χώρος αναζήτησης είναι μη πεπερασμένος, ο DFS μπορεί να μπλεχτεί σε κλαδιά μεγάλου μήκους ή σε ατέρμονα κλαδιά του δέντρου (κλαδιά απείρου μήκους). Συνεπώς, επειδή ο DFS μπορεί να μην βρεί ποτέ μιατελική κατάσταση αν και μπορεί να περάσει από πολύ κοντά της, θεωρείται εν γένει, μη-πλήρης. Στις περιπτώσεις όμως που ο χώρος αναζήτησης είναι πεπερασμένος και χρησιμοποιείται κλειστό σύνολο, ο DFS θα βρεί λύση, εαν μια τέτοια υπάρχει.

Η διαφορά του BFS από τον DFS εντοπίζεται στην περιγραφή του αλγορίθμου, και συγκεκριμένα σε μια μονό λέξη, «τέλος» αντί για «αρχή». Στην περίπτωση του BFS το μέτωπο της αναζήτησης είναι μια δομή ουράς (Queue FIFO, δηλαδή First In First Out) και όχι στοίβας. Έτσι, ποτέ δεν επεκτείνεται μια κατάσταση αν δεν επεκταθούν πρώτα όλες οι καταστάσεις που βρίσκονται σε μικρότερο βάθος, γιατί απλά οι τελευταίες μπήκαν στο μέτωπο της αναζήτησης νωρίτερα.

1.7 Αλγόριθμοι ευριστικής αναζήτησης

Οι αλγόριθμοι τυφλής αναζήτησης προχωρούν σε πλάτος ή σε βάθος χωρίς να έχουν καμία απολύτως πληροφορία για το αν το μονοπάτι που ακολουθούν τους οδηγεί σε κάποια τερματική κατάσταση. Συνεπώς, η τυφλή αναζήτηση δεν επαρκεί για μεγάλους χώρους καταστάσεων που συνήθως εμφανίζονται σε πραγματικά προβλήματα. Ο ρυθμός με τον οποίο αναπτύσσεται ένας χώρος αναζήτησης είναι ταχύτατος με αποτέλεσμα να παρατηρείται το φαινόμενο της συνδιαστικής έκρηξης. Σε τέτοιους χώρους, η τυφλή αναζήτηση διαρκεί τόσο πολύ χρόνο που πρακτικά η λύση δεν βρίσκεται ποτέ.

Σκοπός λοιπόν είναι να μειωθεί ο χρόνος αναζήτησης, δηλαδή ουσιαστικά να μειωθεί ο αριθμός των καταστάσεων που εξετάζει ένας αλγόριθμος. Για να επιτευχθεί κάτι τέτοιο είναι απαραίτητη η ύπαρξη κάποιας πληροφορίας για την αξιολόγηση των καταστάσεων η οποία θα

είναι ικανή να καθοδηγήσει την αναζήτηση σε καταστάσεις που οδηγούν σε μια λύση και ίσως να βοηθήσει στο κλάδεμα ορισμένων καταστάσεων που δεν οδηγούν πουθενά. Οι αλγόριθμοι που εκμεταλεύονται τέτοιες πληροφορίες ονομάζονται *αλγόριθμοι ευριστικής αναζήτησης (heuristic search algorithms)*.

1.7.1 Αλγόριθμος αναρρίχηση λόφου

Η *αναρρίχηση λόφου (Hill-Climbing Search -HC)* είναι ένας αλγόριθμος αναζήτησης που μοιάζει πολύ με τον αλγόριθμο DFS. Ένας αλγόριθμος αναρρίχησης λόφου μπορεί να αναπαρασταθεί με ένα γράφημα, όμως η μέθοδος δεν διατηρεί δένδρο αναζήτησης. Αντίθετα, κάθε κόμβος αποτελεί μία δομή δεδομένων, η οποία περιέχει μόνο την περιγραφή της κατάστασης και την εκτίμηση αυτής. Για κάποιον επιλεγμένο κόμβο, η μέθοδος κατευθύνει την αναζήτηση στον κόμβο, ο οποίος έχει τη μεγαλύτερη τιμή συνάρτησης εκτίμησης μεταξύ των διαδόχων κόμβων. Αν ο διάδοχ ος κόμβος έχει μεγαλύτερη τιμή από τον πρόγονο, τότε η αναζήτηση συνεχίζεται, διαφορετικά η αναζήτηση σταματά και ο πρόγονος κόμβος αποτελεί τη λύση. Έτσι, η μέθοδος αναρριχάται στην επιφάνεια των καταστάσεων μέχρι να φτάσει σε ένα μέγιστο σημείο.

Αναλυτικά η περιγραφή του αλγορίθμου είναι:

1. Η αρχική κατάσταση είναι η τρέχουσα κατάσταση.
2. Αν η κατάσταση είναι μια τελική τότε ανέφερε τη λύση και σταμάτησε.
3. Εφάρμοσε τους τελεστές μετάβασης για να βρείς τις καταστάσεις-παιδιά.
4. Βρές την καλύτερη κατάσταση σύμφωνα με την ευριστική συνάρτηση.
5. Αν η νέα κατάσταση είναι καλύτερη από την τρέχουσα κατάσταση τότε αυτή γίνεται και η τρέχουσα κατάσταση. Πήγαινε στο βήμα 2.
6. Αλλιώς σταμάτα σε αυτήν την κατάσταση (τοπικά καλύτερη).

Οι διαφορές του αλγορίθμου HC με τον DFS είναι οι εξής:

- Στον HC χρησιμοποιείται ευριστική συνάρτηση η οποία καθορίζει ποιά από τις καταστάσεις-παιδιά θα επεκταθεί στη συνέχεια, ενώ στον DFS επιλέγεται μια από τις καταστάσεις-παιδιά, συνήθως η αριστερότερη.

- Στον HC υπάρχει μόνο μια κατάσταση στο μέτωπο αναζήτησης. Κάθε φορά που επιλέγετα μια κατάσταση-παιδί για επέκταση, οι άλλες κλαδεύονται, ενώ στον DFS αποθηκεύονται στο μέτωπο αναζήτησης και εξετάζονται αργότερα μέσω της οπισθοδρόμησης.

Κεφάλαιο 2ο
(Δυναμικός Προγραμματισμός)

2.1. Εισαγωγή

2.1.1 Ιστορία

Ένας αριθμός από διαφορετικών ερευνητών στα οικονομικά και την στατιστική φαίνεται να ανακαλύπτουν ανεξάρτητα την οπισθοδρομική επαγωγή ως τρόπο να λυθούν SDP προβλήματα περιλαμβάνοντας το ρίσκο/την αβεβαιότητα μέσα στη μέση της δεκαετίας του '40. Ο von Neumann και ο Morgenstern (1944) στη εργασία τους πάνω στη θεωρία του παιχνιδιού, χρησιμοποιώντας οπισθοδρομική επαγωγή ανακάλυψαν αυτό που τώρα καλούμε subgame perfect equilibria of extensive form games. Ο Abraham Wald, ο εφευρετης της θεωρίας λήψης αποφάσεων βασισμένη στην στατιστική, επέκτεινε αυτήν την θεωρία στη θεωρία της λήψης διαδοχικών αποφάσεων στο βιβλίο του το 1947 ,*Sequential Analysis*. Ο Wald γενίκευσε το πρόβλημα της καταστροφής του παίκτη από τη θεωρία πιθανότητας και εισήγαγε τη διαδοχική αναλογία δοκιμή πιθανότητας που ελαχιστοποιεί τον αναμενόμενο αριθμό παρατηρήσεων σε μια διαδοχική γενίκευση της κλασσικής δοκιμής υπόθεσης. Εντούτοις ο ρόλος της οπισθοδρομικής επαγωγής είναι λιγότερος προφανής στην εργασία του Wald. Διευκρινίστηκε το 1949 στο έγγραφο Arrow, των Blackwell και Girshick. Μελέτησαν μια γενικευμένη έκδοση προβλήματος λήψης αποφάσεων βασισμένη στην στατιστική, το διατύπωσαν και το έλυσαν με τέτοιο τρόπο ώστε είναι μια εύκολα αναγνωρίσιμη εφαρμογή του σύγχρονου δυναμικού προγραμματισμού. Μετά από τον Wald, αυτοί χαρακτήρισαν τον βέλτιστο κανόνα για μια λήψη απόφασης βασισμένη στην στατιστική, αποτελώντας τις δαπάνες συλλογής των πρόσθετων παρατηρήσεων.

Άλλες πρόωρες εφαρμογές που έκαναν χρήση της οπισθοδρομικής επαγωγής, περιλαμβανομένου της εργασίας του Pierre Mass´e (1944) στη στατιστική υδρολογία και τη διαχείριση των δεξαμενών, και το Arrow, Harris, και την ανάλυση Marschak (1951) της βέλτιστης πολιτικής καταλόγων. Ο Bellman Richard μελέτησε την κοινή δομή που κρύβεται κάτω από SDPs, και μέσα από τις μελέτες του επιδεικνύεται πώς η οπισθοδρομική επαγωγή μπορεί να εφαρμοστεί για να λύσει μια τεράστια κατηγορία SDPs που περιλαμβάνουν αβεβαιότητα. Το μεγαλύτερο μέρος της εργασίας του Bellman σε αυτόν τον τομέα έγινε στην RAND Corporation,, που αρχίζει το 1949. Κατά την διάρκεια της εργασίας του στην επιχείρηση εφεύρε τον ορισμό του δυναμικού προγραμματισμού που είναι τώρα το γενικά αποδεκτό συνώνυμο για την οπισθοδρομική επαγωγή.

2.1.2 Γενικά

Ο Δυναμικός Προγραμματισμός είναι μία υπολογιστική μέθοδος η οποία εφαρμόζεται όταν πρόκειται να ληφθεί μία σύνθετη απόφαση η οποία προκύπτει από τη σύνθεση επιμέρους αποφάσεων που αλληλοεξαρτώνται. Η μέθοδος επίλυσης τέτοιων προβλημάτων βασίζεται στη διασύνδεση των επιμέρους αποφάσεων με κατάλληλη αναδρομική σχέση ώστε η σύνθεση των επιμέρους αποφάσεων να δίνει την τελικά ζητούμενη απόφαση. Το αρχικό πρόβλημα διασπάται σε επιμέρους υποπροβλήματα τα οποία συνδέονται με τη βοήθεια κατάλληλων αναδρομικών σχέσεων.

Για να καλυφθούν όλες οι εκδοχές από τη διασύνδεση των επιμέρους προβλημάτων, τα υποπροβλήματα αυτά λύνονται παραμετρικά, δηλαδή για όλες τις δυνατές τιμές ορισμένων παραμέτρων.

Στο πρώτο κεφάλαιο αναλύεται ο Δυναμικός Προγραμματισμός από άποψη κόστους και αναφέρονται μερικά προβλήματα τα οποία μπορούν να λυθούν με χρήση της συγκεκριμένης μεθόδου.

Χαρακτηριστικό του Δυναμικού Προγραμματισμού είναι ότι δεν υπάρχει γενικευμένη διατύπωση της μεθόδου που να έχει άμεση λειτουργική ισχύ. Οι αναδρομικές σχέσεις που συνεπάγεται η μέθοδος διαφοροποιούνται ριζικά από πρόβλημα σε πρόβλημα. Γι' αυτό και στο δεύτερο κεφάλαιο θα μελετηθούν οι ιδιότητες του Δυναμικού Προγραμματισμού, καθώς και οι δυνατές εφαρμογές του, με τη βοήθεια μερικών χαρακτηριστικών παραδειγμάτων.

2.1.3 Κόστος

Η παραμετρική λύση των υποπροβλημάτων αποτελεί και το κυρίως υπολογιστικό κόστος της μεθόδου, το οποίο, αν και είναι σημαντικό, είναι πάντως πολύ μικρότερο από το κόστος της πλήρους απαρίθμησης και αξιολόγησης όλων των δυνατών λύσεων.

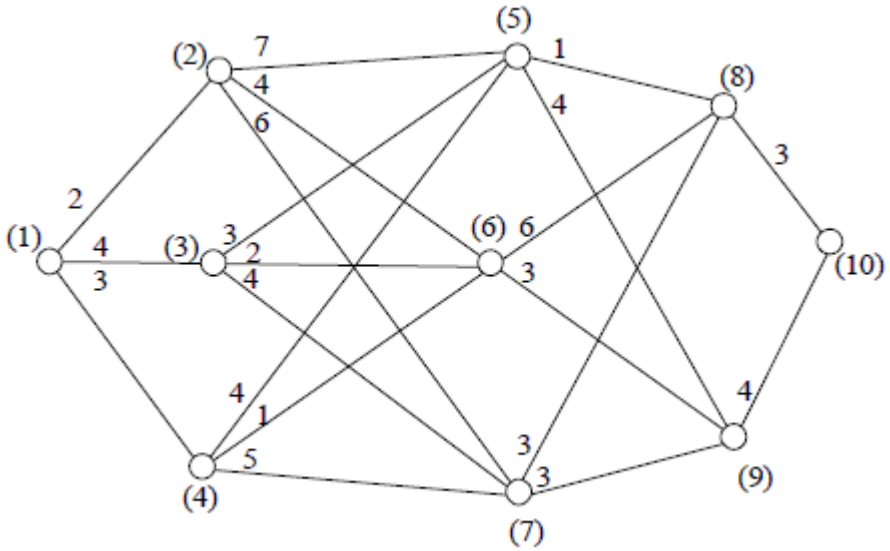
Επειδή το κόστος της υπολογιστικής προσπάθειας στα προβλήματα Δυναμικού Προγραμματισμού είναι αρκετά υψηλό, η μέθοδος χρησιμοποιείται για προβλήματα που δεν είναι δυνατό να αντιμετωπισθούν με μεθόδους Γραμμικού ή Ακέραιου Προγραμματισμού.

Από τη σκοπιά αυτή η μέθοδος Δυναμικός Προγραμματισμός παρουσιάζει μεγαλύτερη ευελιξία από άλλες μεθόδους, το τίμημα όμως για την ευελιξία αυτή είναι συνήθως το αυξημένο υπολογιστικό κόστος.

Παράδειγμα 1ο (υπολογισμός ελάχιστου μονοπατιού)

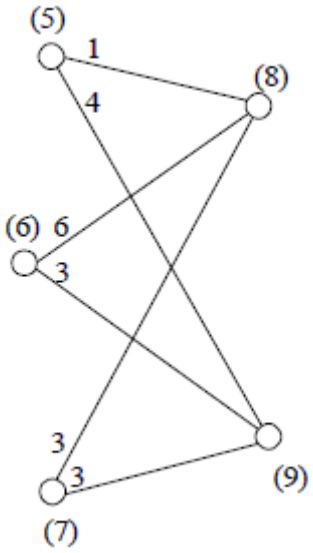
Στο δίκτυο του Σχήματος 2.1 ζητείται να βρεθεί ο συντομότερος δρόμος από τον κόμβο (1) στον κόμβο (10) του δικτύου. Οι αριθμοί στην αρχή κάθε κλάδου παριστάνουν τις επιμέρους αποστάσεις μεταξύ των κόμβων. Αντί να υπολογίσουμε το συνολικό μήκος των (3*3*2=) 18 δυνατών διαδρομών από το (1) στο (10), ένας πιο αποτελεσματικός τρόπος να λύσουμε το πρόβλημα είναι να το "σπάσουμε" σε μικρότερα προβλήματα τα οποία λύνουμε διαδοχικά και συνδέουμε τις λύσεις τους. Δηλαδή, αντιμετωπίζουμε το πρόβλημα σε χωριστά βήματα, όπου καθένα αποτελεί την επίλυση ενός επιμέρους προβλήματος που η λύση του δίνει πληροφορίες για την επίλυση του επόμενου προβλήματος, μέχρις ότου φτάσουμε στο αρχικό πρόβλημα.

ΣΧΗΜΑ 2.1 ΠΑΡΑΔΕΙΓΜΑ ΚΛΕΙΣΤΟΥ ΓΡΑΦΟΥ



Έτσι αρχίζοντας από το τέλος, βρίσκουμε ότι η ελάχιστη διαδρομή από το (8) στο (10) είναι 3 και η ελάχιστη διαδρομή από το (9) στο (10) είναι 4. Οπισθοχωρώντας ένα βήμα ακόμη, βρίσκουμε την ελάχιστη απόσταση από το (5) στο (10) χρησιμοποιώντας τα προηγούμενα αποτελέσματα.

ΣΧΗΜΑ 2.2 ΜΕΡΟΣ ΤΟΥ ΚΛΕΙΣΤΟΥ ΓΡΑΦΟΥ



Η ελάχιστη απόσταση από το (5) στο (10) είναι:
min {1+3, 4+4} = 4

και η διαδρομή είναι μέσω του (8).
Μπορούμε να επεκταθούμε προς τα πίσω, **βήμα-βήμα**. Έτσι έχουμε:
- Η ελάχιστη απόσταση από το (6) στο (10) είναι 7 (μέσω (9)).
- Η ελάχιστη απόσταση από το (7) στο (10) είναι 6 (μέσω (8)).

Βήμα 3 από το τέλος:
- Η ελάχιστη απόσταση από το (2) στο (10) είναι 11 (μέσω (5) ή (6)).
- Η ελάχιστη απόσταση από το (3) στο (10) είναι 7 (μέσω (5)).
- Η ελάχιστη απόσταση από το (4) στο (10) είναι 8 (μέσω (5) ή (6)).

Βήμα 4 από το τέλος:
- Η ελάχιστη απόσταση από το (1) στο (10) είναι 11 (μέσω (3) ή (4)).

Από τα αποτελέσματα αυτά και μόνο μπορούμε να προσδιορίσουμε τη βέλτιστη διαδρομή: Από το (1) η πρώτη απόφαση μας φέρνει στο (3) ή στο (4). Εάν πάμε στο (3), τότε το επόμενο βήμα είναι στο (5), ύστερα στο (8) και τέλος στο (10). Εάν πάμε στο (4), τότε το επόμενο βήμα είναι στο
(5) ή στο (6). Εάν πάμε στο (5), τότε τα επόμενα βήματα είναι (8), (10). Εάν πάμε στο (6), τότε προχωρούμε μέσω (9) στο (10). Έτσι υπάρχουν τρεις βέλτιστες διαδρομές:

(1) - (3) - (5) - (8) - (10)
(1) - (4) - (5) - (8) - (10)
(1) - (4) - (6) - (9) - (10)

Και οι τρεις έχουν, φυσικά, το ίδιο συνολικό μήκος 11.

2.1.4 Χαρακτηριστικά προβλημάτων Δυναμικού Προγραμματισμού: Η αρχή της βελτιστοποίησης του Bellman

Τα προβλήματα Δυναμικού Προγραμματισμού παρουσιάζουν τα ακόλουθα χαρακτηριστικά:

1) Οι αποφάσεις λαμβάνονται διαδοχικά.

2) Το πρόβλημα μπορεί να διαιρεθεί σε βήματα (φάσεις) και σε κάθε βήμα απαιτείται να ληφθεί μια "στρατηγική" απόφαση.

3) Κάθε βήμα έχει ένα ορισμένο αριθμό "καταστάσεων" που συνδέονται με αυτό.

4) Το αποτέλεσμα μιας στρατηγικής απόφασης που λαμβάνεται σε κάθε βήμα είναι να μετατρέπει την παρούσα κατάσταση σε μια κατάσταση που συνδέεται με το επόμενο βήμα.

5) Με κάθε απόφαση συνδέεται ένα κέρδος ή μία ζημία (κόστος).Για παράδειγμα, στο προηγούμενο πρόβλημα το 1ο βήμα από το τέλος είχε τις δυνατές καταστάσεις (8), (9), ενώ το 2ο βήμα από το τέλος τις καταστάσεις (5), (6) και (7). Εάν βρισκόμαστε σε μια από τις καταστάσεις του 2ου βήματος από το τέλος, η απόφαση να ακολουθήσουμε ένα ορισμένο κλάδο μας φέρνει σε μια από τις καταστάσεις του επόμενου βήματος (1ο βήμα από το τέλος).

6) Ο αντικειμενικός σκοπός, που εκφράζεται από την αντικειμενική συνάρτηση, είναι να μεγιστοποιηθεί το συνολικό κέρδος ή να ελαχιστοποιηθεί η συνολική ζημία, ή γενικότερα να επιτευχθεί το καλύτερο δυνατό αποτέλεσμα.

7) Τέλος, ο τρόπος με τον οποίο βρεθήκαμε σε μια κατάσταση ενός βήματος είναι άσχετος με τις αποφάσεις που θα επακολουθήσουν. Δηλαδή οι αποφάσεις που θα επακολουθήσουν εξαρτώνται μόνο από την κατάσταση στην οποία βρισκόμαστε και όχι από τον τρόπο με τον οποίο βρεθήκαμε σ' αυτήν την κατάσταση.

Τέτοια προβλήματα διέπονται από την ακόλουθη "αρχή του Bellman" που χαρακτηρίζει τη βέλτιστη λύση: **"Μια βέλτιστη διαδοχή αποφάσεων έχει την ιδιότητα ότι, ανεξάρτητα από τις αρχικές αποφάσεις, οι αποφάσεις που απομένουν πρέπει να συνιστούν μια βέλτιστη στρατηγική (πολιτική) σε σχέση με την κατάσταση που απορρέει από τις αρχικές αποφάσεις"**.

Στο παράδειγμα που ήδη εξετάστηκε, εάν υποθέσουμε ότι μια διαδοχή αποφάσεων είναι βέλτιστη και ότι οι δύο πρώτες αποφάσεις (αρχικές) αυτής της διαδοχής μας φέρνουν στον κόμβο (5), θα πρέπει και η υπόλοιπη διαδρομή, από τον κόμβο (5) στον (10), να είναι βέλτιστη, δηλαδή θα πρέπει και οι υπόλοιπες αποφάσεις να δίνουν τη συντομότερη διαδρομή από τον κόμβο (5) στον (10).

2.2. Η γενική αναδρομική σχέση

Μπορούμε να εκφράσουμε φορμαλιστικά τις παραπάνω ιδιότητες χρησιμοποιώντας ορισμένους συμβολισμούς. Έστω:

n : ο αριθμός των φάσεων που απομένουν, δηλαδή n είναι ο δείκτης αρίθμησης των φάσεων αρχίζοντας από το τέλος.

x_n : η μεταβλητή που καθορίζει την απόφαση στη φάση n (από το τέλος).

s : η μεταβλητή που καθορίζει την κατάσταση που βρισκόμαστε.

$f_n(s,x_n)$: η συνάρτηση που εκφράζει το βέλτιστο αποτέλεσμα για τις n τελευταίες φάσεις μαζί, όταν στη n-οστή - από το τέλος - βρισκόμαστε στην κατάσταση s και παίρνουμε την απόφαση που καθορίζει η μεταβλητή x_n.

$r(x_n,s)$: το κέρδος ή ζημία που προκύπτει όταν βρισκόμαστε στην κατάσταση s της n-οστής φάσης και πάρουμε την απόφαση x_n.

$T(x_n,s)$: η κατάσταση της φάσης n-1 στην οποία μας οδηγεί η απόφαση x_n που λαμβάνεται όταν βρισκόμαστε στην κατάσταση s της n-οστής φάσης.

Τότε ισχύει η αναδρομική σχέση:

$$f_n(s) = \max_{x_n}\{r_n(x_n, s) + f_{n-1}(T(x_n, s)\}$$

Έστω $\overline{X}_n$ η τιμή της x_n που δίνει τη βέλτιστη τιμή της $f_n(s)$. Τότε έχουμε:

$$f_n(s) = \max_{x_n}(f\ (x_n, s) = f(\overline{x}_n, s)$$

Συνήθως οι τιμές $f_1(s)$ για τα διάφορα s είναι εύκολο να βρεθούν. Έτσι μπορούμε να εφαρμόσουμε αναδρομικά τη σχέση που βρήκαμε και να επωφεληθούμε από το γεγονός ότι η τιμή του s είναι συνήθως καθορισμένη για την αρχική φάση (την πρώτη από την αρχή). Οι τιμές $\overline{x}_n$ μας δίνουν τη διαδοχή των αποφάσεων που οδηγούν στη βέλτιστη λύση.

Παράδειγμα 2ο

Μια επιχείρηση επιθυμεί να κατανείμει 5 μονάδες προϊόντος στα τέσσερα καταστήματά της ώστε να μεγιστοποιήσει τα κέρδη της. Τα αναμενόμενα κέρδη από την κατανομή των μονάδων στα διαφορετικά καταστήματα δίνονται από τον παρακάτω πίνακα:

ΠΙΝΑΚΑΣ 2.1 ΠΑΡΑΔΕΙΓΜΑ ΚΑΤΑΣΤΗΜΑΤΩΝ-ΜΕΓΙΣΤΟΠΟΙΗΣΗΣ ΚΕΡΔΟΥΣ

ΜΟΝΑΔΕΣ	ΚΑΤΑΣΤΗΜΑΤΑ			
	Α	Β	Γ	Δ
0	0	0	0	0
1	1	2	2	3
2	2	4	3	4
3	3	5	4	4
4	4	5	5	4

Υποτίθεται ότι δεν προκύπτει επιπλέον κέδρος με το να κατανείμουμε περισσότερες από 4 μονάδες σ' ένα μόνο κατάστημα. Έστω:

$f_n(s)$: το μέγιστο δυνατό κέρδος που προκύπτει από την κατανομή s μονάδων στα τελευταία n καταστήματα.

x_n : ο αριθμός των μονάδων που κατανέμονται στο n-οστό από το τέλος κατάστημα. Δηλαδή εδώ το κάθε βήμα αντιστοιχεί στην κατανομή μονάδων σ' ένα από τα καταστήματα.

Έστω $\overline{X}_n$ το βέλτιστο x_n για κάποιο ορισμένο s.

Τότε:

s	0	1	2	3	4	5
$f_1(s)$	0	3	4	4	4	4
$\overline{x}_1$	0	1	2	2,3	2,3,4	2,3,4,5

και

Σφάλμα! Δεν έχει οριστεί σελιδοδείκτης.

$$f_n(s) = \max_{0 \le xn \le s} \{r_n(x_n) + f_{n-1}(s - x_n)\}$$

όπου $r_n(x_n)$, το κέδρος που προκύπτει από την κατανομή x_n μονάδων στο n-οστό (από το τέλος) κατάστημα. Προφανώς, εάν διατίθενται συνολικά s μονάδες για τα τελευταία n καταστήματα και κατανεμηθούν x_n στο n-οστό, θα μείνουν $s-x_n$ για τα τελευταία n-1. Έτσι όταν n=2 έχουμε για το $f_2(s)$:

$f_2(x_2, s) = [r_2(x_2) + f_1(s - x_2)]$

	x_2							
s	0	1	2	3	4	5	$f_2(s)$	$\overline{x}_2$
0	0+0=0	–	–	–	–	–	0	0
1	0+3=3	2+0=2	–	–	–	–	3	0
2	0+4=4	2+3=5	3+0=3	–	–	–	5	1
3	0+4=4	2+4=6	3+3=6	4+0=4	–	–	6	1,2
4	0+4=4	2+4=6	3+4=7	4+3=7	5+0=5	–	7	2,3
5	0+4=4	2+4=6	3+4=7	4+4=8	5+3=8	5+0=5	8	3,4

Παρόμοια για n=3 ο συνοπτικός πίνακας που δίνει το f3(s) και το αντίστοιχο $\bar{x}_3$ είναι (οι αναλυτικοί υπολογισμοί παραλείπονται):

s	0	1	2	3	4	5
$f_3(s)$	0	3	5	7	9	10
$\bar{x}_3$	0	0	0,1	1,2	2	2,3

Τέλος για n=4 έχουμε:

$f_4(s) = \max\{0+10, 1+9, 2+7, 3+5, 4+3, 4+0\} = 10$ με $\bar{x}_4 = 0,1$.

Έτσι προκύπτουν οι βέλτιστες κατανομές:

ΠΙΝΑΚΑΣ 2.2 ΒΕΛΤΙΣΤΕΣ ΚΑΤΑΝΟΜΕΣ ΠΑΡΑΔΕΙΓΜΑΤΟΣ ΚΑΤΑΣΤΗΜΑΤΩΝ

Βέλτιστες Κατανομές	ΚΑΤΑΣΤΗΜΑΤΑ			
$1^{η}$	Α	Β	Γ	Δ
$2^{η}$	0	2	1	2
$3^{η}$	0	2	2	1
$4^{η}$	0	3	1	1
$5^{η}$	1	2	1	1

2.3 Δυναμικός προγραμματισμός και αβεβαιότητα

Με την παρουσία αβεβαιότητας η κάθε απόφαση μπορεί να έχει περισσότερα από ένα "δυνατά" αποτελέσματα, το καθένα από τα οποία έχει μια καθορισμένη πιθανότητα να συμβεί και, επίσης, μπορεί να οδηγήσει σε μετασχηματισμό της παρούσας "κατάστασης" σε περισσότερες από μια
δυνατές καταστάσεις της επόμενης φάσης, η καθεμιά από τις οποίες έχει μια ορισμένη πιθανότητα να συμβεί.

Παράδειγμα 3ο

Ένας παίκτης αρχίζει το παιχνίδι του με δύο μάρκες. Πρόκειται να στοιχηματίσει 3 φορές και κάθε φορά μπορεί να παίξει το πολύ όσες μάρκες έχει στη διάθεσή του. Σε κάθε στοίχημα (γύρο) είτε κερδίζει είτε χάνει τόσες μάρκες όσες στοιχηματίζει. Σε κάθε γύρο η πιθανότητα επιτυχίας είναι 0,6 , και η πιθανότητα αποτυχίας 0,4. Ο παίκτης ενδιαφέρεται να μεγιστοποιήσει την πιθανότητα στο τέλος των τριών γύρων να καταλήξει με τουλάχιστον 4 μάρκες, ώστε να μπορεί να πληρώσει το ξενοδοχείο του.

Έστω $f_n(s)$ η πιθανότητα ότι, με n γύρους ακόμη να παίξει και με s μάρκες στην κατοχή του, ο παίκτης μπορεί να κατορθώσει να τελειώσει το παιγνίδι έχοντας στην κατοχή του συνολικά r μάρκες (r=4). Έστω xn οι μάρκες που στοιχηματίζει στο n-οστό γύρο από το τέλος, τότε:

$$f_n(s) = \max_{0 \le xn \le s} \{0.6 * f_{n-1}(s + x_n) + 0{,}4 * f_{n-1}(s - x_n)\}$$

Για n=1 έχουμε:

s	$f_1(s)$	$\bar{x}_1$
0	0	0
1	0	0,1
2	0,6	2
3	0,6	1,2,3
≥4	1	≤(s−4)

Για n=2 έχουμε:

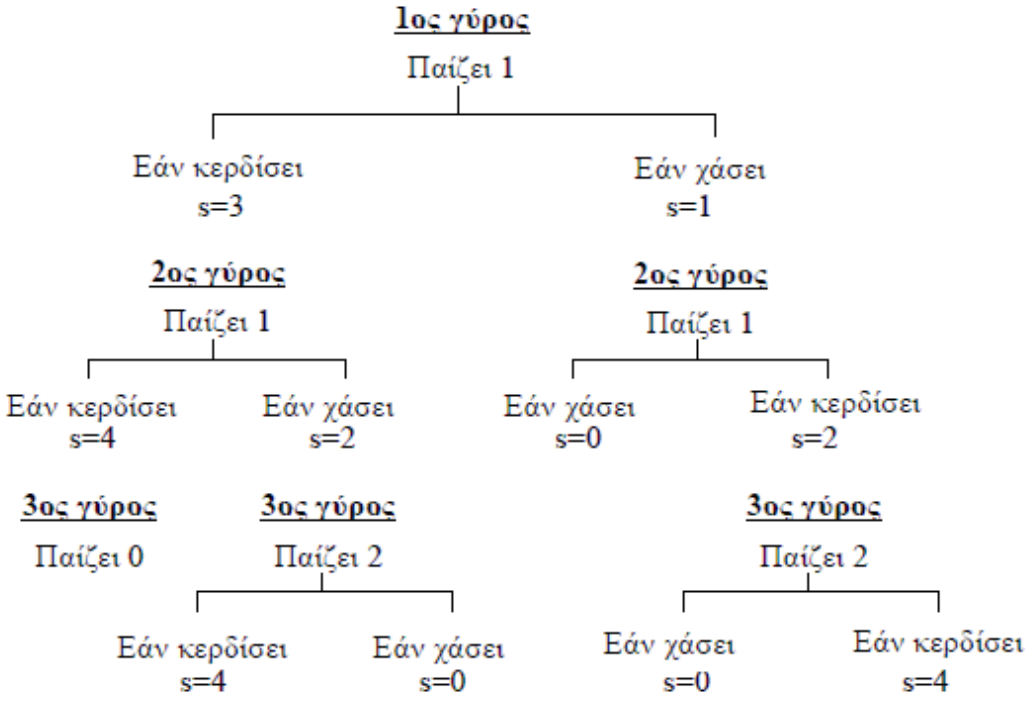
	x_2				$f_2(s)$	$\bar{x}_2$
s	0	1	2	3		
0	$f_1(0)$=0	–	–	–	0	0
1	$f_1(1)$=0	0,6×0,6+0,4×0=0,36	–	–	0,36	1
2	$f_1(2)$=0,6	0,6×0,6+0,4×0=0,36	0,6×1+0,4×0=0,6	–	0,6	0,2
3	$f_1(3)$=0,6	0,6×1+0,4×0,6=0,84	0,6×1+0,4×0=0,6		0,84	1
4	$f_1(4)$=1	0,6×1+0,4×0,6<1	<1	<1	1	0
>4					1	≤(s−4)

Τέλος για n=3 και s=2 έχουμε:

	x_3			$f_3(2)$	$\bar{x}_3$
s	0	1	2		
2	$f_2(2)=0,6$	0,6×0,84+0,4×0,36=0,648	0,6×1+0,4×0=0,6	0,648	1

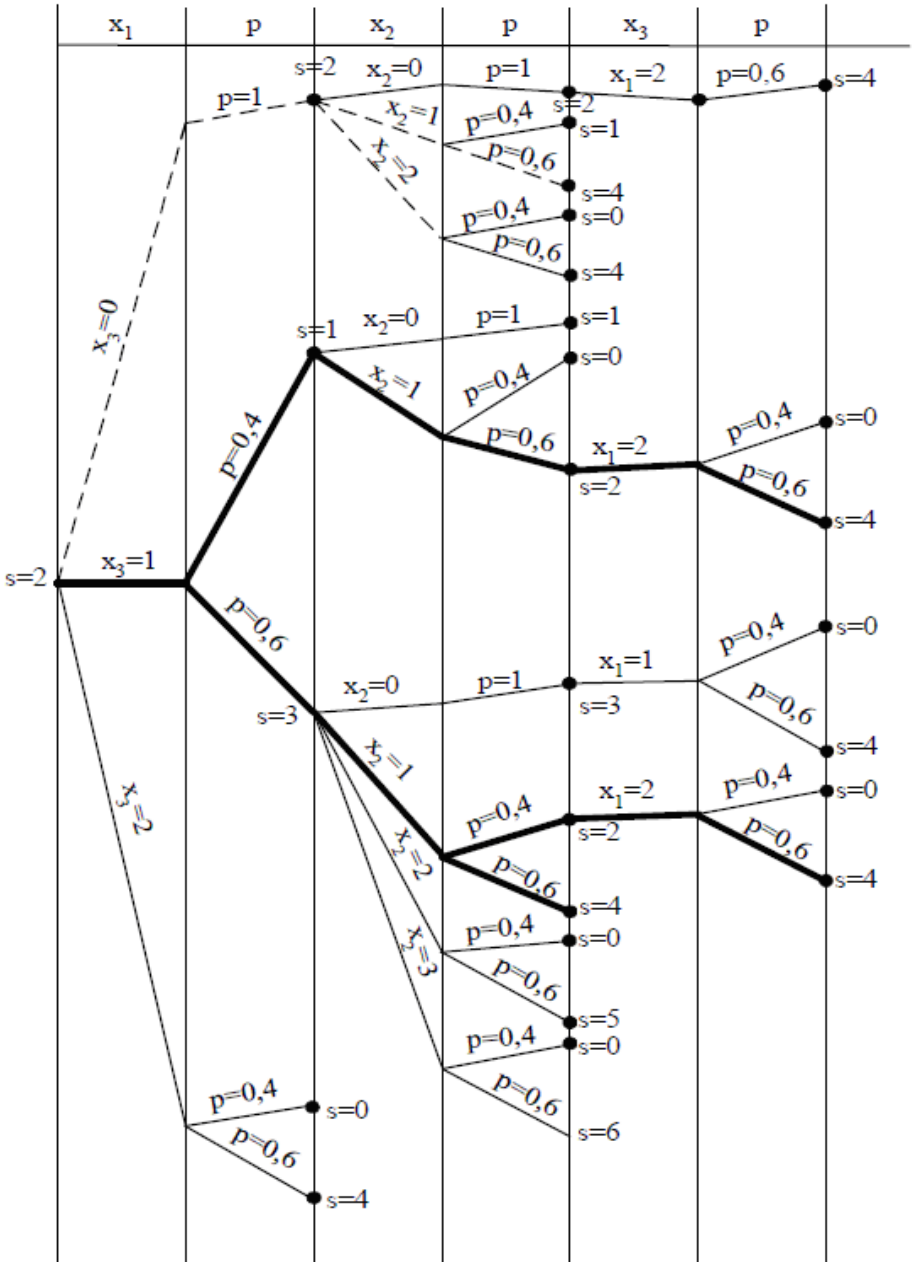
Έτσι η λύση του παιχνιδιού είναι:

ΣΧΗΜΑ 2.3 ΑΝΑΠΤΥΞΗ ΛΥΣΗΣ ΠΑΙΧΝΙΔΙΟΥ ΜΕ ΚΕΡΜΑΤΑ



Στο Σχήμα 2.3 παριστάνονται γραφικά όλες οι δυνατές επιλογές του παίκτη. Ορισμένες επιλογές δεν συνεχίζονται στα επόμενα βήματα. Στις περιπτώσεις αυτές η επιτυχία ή αποτυχία αντίστοιχα είναι εξασφαλισμένη ήδη, πράγμα που σημαίνει ότι η πιθανότητα επιτυχίας με κατάλληλη στρατηγική είναι 1 ή αντίστοιχα με οποιαδήποτε στρατηγική η αποτυχία είναι δεδομένη. Για την άριστη στρατηγική ο παίκτης ξεκινά με $x_3=1$ οπότε η πιθανότητα επιτυχίας είναι: 0,4*0,6*0,6 + 0,6*[0,4*0,6*0,6] = 0,144+0,504=0,648. Αυτή αντιστοιχεί σε $x_3=1$.Εύκολα επαληθεύεται ότι κάθε άλλη στρατηγική έχει μικρότερη πιθανότητα επιτυχίας.

ΣΧΗΜΑ 2.4 ΠΛΗΡΗΣ ΑΝΑΠΤΥΞΗ ΛΥΣΗΣ ΠΑΙΧΝΙΔΙΟΥ ΜΕ ΚΕΡΜΑΤΑ



2.4 Διαφορές Δυναμικού Προγραμματισμού – "Διαίρει και Βασίλευε"

Όπως και η μέθοδος "Διαίρει και Βασίλευε" έτσι και ο δυναμικός προγραμματισμός επιλύει προβλήματα συνδυάζοντας λύσεις υποπροβλήματων.

Όμως μια σημαντική διαφορά των δύο μεθόδων είναι ότι η μέθοδος "διαίρει και βασίλευε" χωρίζει ένα πρόβλημα σε ανεξάρτητα υποπροβλήματα, λύνει αυτά τα προβλήματα αναδρομικά και συνδυάζει τις λύσεις για να λύσει το αρχικό πρόβλημα. Αντίθετα ο Δυναμικός Προγραμματισμός είναι εφαρμόσιμος και εκεί όπου τα υποπροβλήματα "επικαλύπτονται", και έχουν κοινά υποπροβλήματα. Ενώ ένας "διαίρει και βασίλευε" αλγόριθμος θα έλυνε κοινά υποπροβλήματα ξανά και ξανά, ένας αλγόριθμος Δυναμικού Προγραμματισμού λύνει κάθε υποπρόβλημα ακριβώς μια φορά και καταχωρεί τη λύση σε ένα πίνακα από όπου μπορεί να την διαβάσει κάθε φορά που το πρόβλημα ξανασυναντιέται.

2.5 Πλεονεκτήματα και Μειονεκτήματα της Μεθόδου του Δυναμικού Προγραμματισμού

- Είναι ιδανική για προγραμματισμό μέσω υπολογιστή.
- Είναι εφαρμόσιμη σε μια απεριόριστη κλάση προβλημάτων βελτιστοποίησης (μη γραμμικές, χρονικά μεταβαλλόμενες εξισώσεις κατάστασης, αρκετά πολύπλοκος δείκτης απόδοσης). Ακόμα και σε προβλήματα στοχαστικού βέλτιστου ελέγχου.
- Χειρίζεται περιορισμούς γενικής φύσης, οι οποίοι στην ουσία απλοποιούν τη μέθοδο.
- Καταλήγει σε βέλτιστο έλεγχο κλειστού βρόχου. Δυνατότητα off-line υπολογισμών.
- Εγγυάται για το ολικό ελάχιστο.
- Μειωμένο υπολογιστικό κόστος σε σχέση με αυτό της μεθόδου άμεσης απαρίθμησης.

  Για π.χ. έστω ένα βαθμωτό σύστημα με μια μεταβλητή ελέγχου. Δοκιμάζουμε από 5 τιμές ελέγχου και 10 τιμές κατάστασης. Τότε μέσω του δυναμικού
  προγραμματισμού, το πλήθος των απαιτούμενων υπολογισμών είναι ίσο με **50N** (γραμμική), ενώ μέσω της άμεσης απαρίθμησης είναι ίσο με (εκθετική).

- Η «κατάρα της διαστατικότητας» αποτελεί τον πιο περιοριστικό παράγοντα στην εξάπλωση της μεθόδου. Υπερβολικές υπολογιστικές απαιτήσεις, ακόμα και για μικρής τάξης συστήματα.
- Δεν είναι δυνατή πάντα η εύρεση μιας αναλυτικής λύσης.

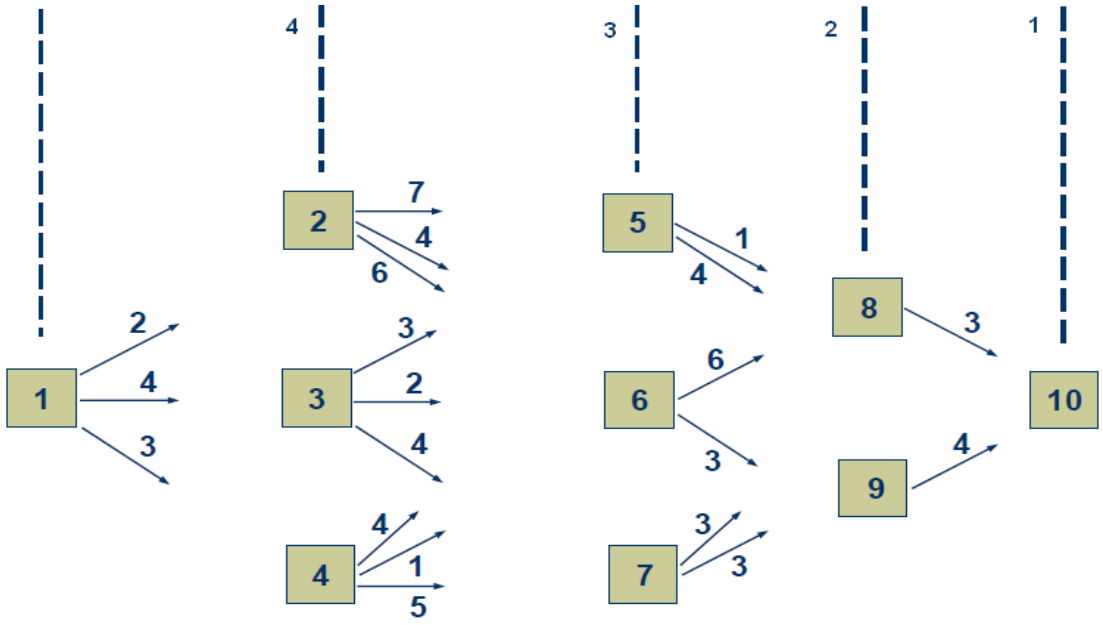
Κεφάλαιο 3^{ο}
(Εφαρμογές)

Η βασική δυσκολία του δυναμικού προγραμματισμού έγκειται στο γεγονός ότι δεν υπάρχει καμία μαγική συνταγή για να βρίσκουμε την βέλτιστη λύση (αν υπάρχει) ενός προβλήματος. Ο δρόμος έρχεται μέσα από την διαρκή εξάσκηση και μελέτη, των ήδη υπάρχοντων αλγορίθμων, ώστε να αναπτυχθεί η σχετική οικειότητα με το αντικείμενο.

Στο κεφάλαιο αυτό θα παρουσιάσουμε με όσο το δυνατόν πιο απλό και κατανοητό τρόπο μερικές εφαρμογές του δυναμικού προγραμματισμού, σε αντίστοιχα προβλήματα, έτσι ώστε να υπάρξει μια άμεση και πρακτική επαφή με την τεχνική.

3.1 Βέλτιστη Διαδρομή

ΣΧΗΜΑ 3.1 ΠΑΡΑΔΕΙΓΜΑ ΚΛΕΙΣΤΟΥ ΓΡΑΦΟΥ



Έστω ότι υπάρχει ο παραπάνω γράφος και χρειάζεται να βρεθεί το ελάχιστο μονοπάτι. Χωρίζεται σε 4 στάδια (n=1,2,3,4) μετρώντας από το τέλος, σαν μεταβλητή απόφασης X_n είναι ο αμέσως επόμενος προορισμός ,σαν μεταβλητή κατάστασης S_n είναι ο κόμβος στον οποίο βρισκόμαστε και χρησιμοποιώντας την αναδρομική σχέση

Σφάλμα! Δεν έχει οριστεί σελιδοδείκτης.$F_n(S_n,X_n) = C_{SnXn} + F^*_{n-1}(X_n)$

$F^*_n(S_n) = \min \{ F_n(S_n,X_n) \}$

Όπου C_{SnXn} είναι το κόστος για την διαδρομή $S_n \rightarrow X_n$.

Πρέπει να σημειωθεί ότι το κόστος για τα τελευταία n στάδια θα ισούται με το κόστος της βέλτιστης πολιτικής για τα τελευταία n-1 στάδια (F^*_{n-1}) συν το κόστος της διαδρομής $X_n \rightarrow S_n$ και η σχέση μεταξύ S_n και X_n είναι ο προορισμός του σταδίου n είναι η αφετηρία του σταδίου n-1 δηλαδή $X_n = S_{n-1}$.

Έτσι έχουμε ότι

Για n=1

S_1	8	9
$F^*_1(S_1)$	3	4
X^*_1	10	10

Για n=2 και κάνοντας χρήση της αναδρομικής σχέσης που τυπώθηκε παραπάνω δηλαδή $F_2(S_2,X_2) = C_{S2X2} + F^*_1(X_2)$

S_2 \ X_2	8	9	$F^*_2(S_2)$	X^*_2
5	4	8	4	8
6	9	7	7	9
7	6	7	6	8

Για n=3 και $F_3(S_3,X_3) = C_{S3X3} + F^*_2(X_3)$

S_3 \ X_3	5	6	7	$F^*_3(S_3)$	X^*_3
2	11	11	12	11	5 ή 6
3	7	9	10	7	5
4	8	8	11	8	5 ή 6

Για n=4 και $F_4(S_4,X_4) = C_{S4X4} + F^*_3(X_4)$

S_4 \ X_4	2	3	4	$F^*_4(S_4)$	X^*_4
1	13	11	11	11	3 ή 4

Άρα από τον παραπάνω πίνακα προκύπτει ότι το συνολικό κόστος είναι 11 [$\mathbf{F^*_4(S_4)=11}$] και για να φτάσουμε σε αυτό το αποτέλεσμα ακολουθήσαμε την διαδρομή (δηλαδή η βέλτιστη λύση είναι)

$1 \rightarrow 3 \rightarrow 5 \rightarrow 8 \rightarrow 10$
$1 \rightarrow 4 \rightarrow 5 \rightarrow 8 \rightarrow 10$
$1 \rightarrow 4 \rightarrow 6 \rightarrow 9 \rightarrow 10$

3.2 Πρόβλημα σάκου

Στο πρόβλημα του σάκου θέλουμε να γεμίσουμε ένα σάκο που έχει χωρητικότητα W. Από μία λίστα με n αντικείμενα πρέπει να επιλέξουμε τα αντικείμενα που πρόκειται να τοποθετηθούν στο σάκο. Κάθε αντικείμενο έχει ένα βάρος w_i και ένα κέρδος p_i. Σε μια εφικτή πλήρωση του σάκου το άθροισμα των βαρών των αντικειμένων που έχουν εισαχθεί δεν ξεπερνά τη χωρητικότητα του σάκου. Μία βέλτιστη πλήρωση επιτυγχάνει το βέλτιστο κέρδος.

$$\max\left(\sum_{i=1}^{n} p_i x_i\right) \text{ με } x_i \in [0,1],\ 1 \le i \le n$$

και

$$\sum_{i=1}^{n} w_i x_i \le c$$

Για την επίλυση του προβλήματος πρέπει να τονίσουμε ότι θα λυθεί με χρήση πίνακα όπου αποθηκεύονται τα αποτελέσματα (το μέγιστο κέρδος από τα περιεχόμενα του σακιδίου) σε κάθε στάδιο των αποφάσεων. Κατασκευάζουμε λοιπόν ένα δυσδιάστατο πίνακα V[0…n,0…W]. Για $1 \le i \le n$ και $0 \le w \le W$,το στοιχείο V[i,w] προκύπτει από το προηγούμενο, αυξάνοντας τον χώρο των δεδομένων εισόδου κατά ένα στοιχείο και αυξάνοντας διαδοχικά το συνολικό επιτρεπτό βάρος μέχρι το W.

Η λύση του προβλήματος μπορεί να υπολογισθεί είτε με αναδρομή είτε με επανάληψη. Στην επαναληπτική λύση ξεκινάμε με το v[i,*], όπως δίνεται από την αρχική σχέση και μετά παίρνουμε το V[i,*] με τη σειρά i=n-1, n-2, …,n χρησιμοποιώντας την αναδρομική εξίσωση. Αν όμως επιλεχθεί η χρήση αναδρομής τότε πρέπει να προσέξουμε ώστε να αποφεύγεται ο επαναυπολογισμός ήδη υπολογισμένων τιμών γιατί τότε ο υπολογισμός της λύσης γίνεται εξαιρετικά πολύπλοκος.

Για να βρεθεί η λύση επιλέγουμε να χρησιμοποιήσουμε την επανάληψη. Κατ' επανάληψη καθορίζεται η αξία μιας βέλτιστης λύσης λαμβάνοντας υπόψη μας τις λύσεις των μικρότερων προβλημάτων. Αρχικά ισχύει ότι V[0,w]=0 για $0 \leq w \leq W$ όταν δεν έχει καθόλου αντικείμενα ο σάκος και V[i,w]= $-\infty$ για w<0. Η παραπάνω σχέση με βάση την επαναληπτικότητα μετατρέπεται σε

$$V[i,w] = \max\{V[i-1,w] , u_i + V[i-1,w-w_i]\} \quad \text{για } 1 \leq i \leq n \text{ και } 0 \leq w \leq W$$

Και αυτό ισχύει γιατί για να υπολογιστεί το V[i,w] πρέπει να σημειωθεί ότι έχουμε μόνο 2 επιλογές για κάθε αντικείμενο i.

- ✓ *Αν δεν αποθηκεύσουμε το αντικείμενο i* το καλύτερο που μπορεί να γίνει για τα αντικείμενα {1,2,…,i-1} και με όριο χωρητικότητας w είναι V[i-1,w].
- ✓ *Αν αποθηκεύσουμε το αντικείμενο i* (μόνο αν ισχύει ότι $w_i \leq w$) τότε κερδίζουμε p_i και ξοδεύουμε w_i από τον αποθηκευτικό μας χώρο. Το καλύτερο που μπορεί να γίνει με τα υπόλοιπα αντικείμενα {1,2,…,i-1} και με αποθηκευτικό χώρο ($w - w_i$) είναι $V[i-1,w-w_i]$. Τελικά αποθηκεύουμε $p_i + V[i-1,w-w_i]$.

Επομένως φτάνουμε στους παρακάτω τύπους

για $V[0,*]=0$

και $$V[i,w] = \begin{cases} \max\{V[i-1,w] , u_i + V[i-1,w-w_i]\} & \text{για } 1 \leq i \leq n \text{ και } w_i \leq w \\ V[i-1,w] & \text{για } w < w_i \end{cases}$$

ΠΑΡΑΔΕΙΓΜΑ

Έχουμε 4 αντικείμενα με βάρος w_i=[2,1,3,2] και με κέρδος p_i=[12,10,20,15] και έχουμε στην διάθεση μας αποθηκευτικό χώρο W=5.

ΠΙΝΑΚΑΣ 3.1 ΠΑΡΑΔΕΙΓΜΑ ΠΡΟΒΛΗΜΑΤΟΣ ΣΑΚΟΥ

Ειδος	w_i	p_i
1	2	12
2	1	10
3	3	20

4	2	15

Για i=1 (με w_1=2 και p_2=12)

w	**V[i,j]**
0	0
1	0
2	max{V[0,2],12+V[0,0]}=max{0,12+0}=12
3	max{V[0,3],12+V[0,1]}=max{0,12+0}=12
4	max{V[0,4],12+V[0,2]}=max{0,12+0}=12
5	max{V[0,5],12+V[0,3]}=max{0,12+0}=12

Για i=2 (με w_2=1 και p_2=10)

w	**V[i,j]**
0	0
1	max{V[1,1],10+V[1,0]}=max{0,10+0}=10
2	max{V[1,2],10+V[1,1]}=max{12,10+0}=12
3	max{V[1,3],10+V[1,2]}=max{12,10+12}=22
4	max{V[1,4],10+V[1,3]}=max{12,10+12}=22
5	max{V[1,5],10+V[1,4]}=max{12,10+12}=22

Για i=3 (με w_3=3 και p_3=20)

w	**V[i,j]**
0	0
1	10
2	12
3	max{V[2,3],20+V[2,0]}=max{22,20+0}=22
4	max{V[2,4],20+V[2,1]}=max{22,20+10}=30
5	max{V[2,5],20+V[2,2]}=max{22,20+12}=32

Για i=4 (με w_4=2 και p_4=15)

w	**V[i,j]**
0	0
1	10

2	max{V[3,2],15+V[3,0]}=max{12,15+0}=15
3	max{V[3,3],15+V[3,1]}=max{22,15+10}=25
4	max{V[3,4],15+V[3,2]}=max{30,15+12}=30
5	max{V[3,5],15+V[3,3]}=max{32,15+22}=37

Αν συνοψίσουμε όλα τα παραπάνω αποτελέσματα σε ένα πίνακα προκύπτει ο παρακάτω πίνακας.

ΠΙΝΑΚΑΣ 3.2 ΑΝΑΛΥΤΙΚΟΣ ΠΙΝΑΚΑΣ ΑΠΟΤΕΛΕΣΜΑΤΩΝ

i \ W	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

Άρα βλέπουμε ότι η βέλτιστη λύση είναι τα περιεχόμενα του τελευταίου στοιχείου του πίνακα δηλαδή V[4,5]= 37.

3.3 Το πρόβλημα του πλανόδιου πωλητή

Στο πρόβλημα του πλανόδιου πωλητή μας δίνεται ένας γράφος με βάρη και το ζητούμενο είναι ένας κύκλος ελάχιστου κόστους που να περνάει μια ακριβώς φορά από κάθε κόμβο και να καταλήγει στον κόμβο από τον οποίο ξεκίνησε.

Μπορούμε να δούμε ότι ισχύει η αρχή της βελτιστοποίησης αφού αν διαλέξουμε σαν αρχή τον κόμβο 1 έχουμε ότι

Σφάλμα! Δεν έχει οριστεί σελιδοδείκτης. $MinCostTour_{1\to 1} = \min_{k\neq 1}\{\cos t[1,k] + MinCostTour_{k\to 1}\}$

Ορίζουμε g(i,S) να είναι το κόστος του συντομότερου μονοπατιού από τον κόμβο i στον κόμβο 1, που περνά από όλους τους κόμβους του S. Τότε

$$g(1, V-\{1\}) = \min_{2\le k\le n}\{\cos t[1,k] + g(k, V-\{1,k\})\}$$

Και γενικά ισχύει ότι

$$g(i,S) = \min_{j \in S}\{\cos t[i,j] + g(j, S - \{j\})\}$$

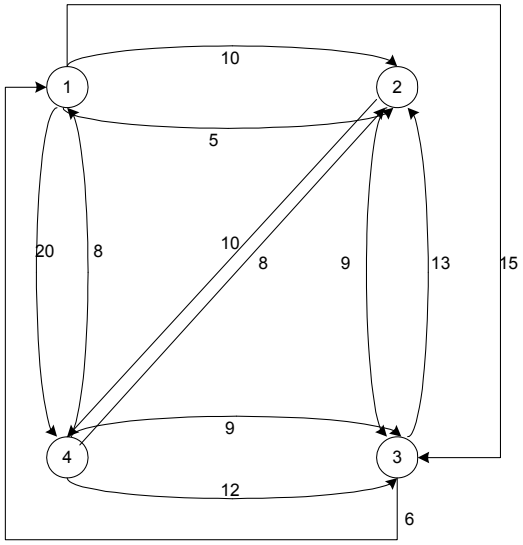
Επίσης g(i,ø)=cost[i,1]

ΠΑΡΑΔΕΙΓΜΑ

Έστω κατευθυνόμενος γράφος με τα παρακάτω στοιχεία

ΣΧΗΜΑ 3.2 ΠΑΡΑΔΕΙΓΜΑ ΚΑΤΕΥΘΥΝΟΜΕΝΟΥ ΓΡΑΦΟΥ

$$C = \begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$



Είναι φανερό ότι αν υπολογίσουμε το g(1,{2,3,4}), το πρόβλημα θα έχει λυθεί. Αρχίζουμε τον υπολογισμό των διαφόρων g για |V|=0, |V|=1 , |V|=2, |V|=3.

Στην περίπτωση που |V|=0 έχουμε ότι

g(2,ø)= Cost[2,1]=5
g(3,ø)= Cost[3,1]=6
g(4,ø)= Cost[4,1]=8

Στην περίπτωση που |V|=1 έχουμε ότι

g(2,{3})=Cost[2,3]+g(3,ø)=9+6=15
g(2,{4})=Cost[2,4]+g(4,ø)=10+8=18
g(3,{2})=Cost[3,2]+g(2,ø)=13+5=18

g(3,{4})=Cost[3,4]+g(4,ø)=12+8=20
g(4,{2})=Cost[4,2]+g(2,ø)=8+5=13
g(4,{3})=Cost[4,3]+g(3,ø)=9+6=15

Στην περίπτωση που |V|=2 έχουμε ότι

g(2,{3,4*})=min(Cost[2,3]+g(3,{4}), Cost[2,4]+g(4,{3})=min(9+20, 10+15)
=25

g(3,{2,4*})=min(Cost[3,2]+g(2,{4}), Cost[3,4]+g(4,{2})=min(13+18, 12+13)
=25

g(4,{2*,3})=min(Cost[4,2]+g(2,{3}), Cost[4,3]+g(3,{2})=min(8+15, 9+18)
=23

Στην περίπτωση που |V|=2 έχουμε ότι

g(1,{2*,3,4})=min(Cost[1,2]+g(2,{3,4}). Cost[1,3]+g(3,{2,4}),
Cost[1,4]+g(4,{2,3}))
=min(10+25, 15+25, 20+23)=35

Τα αστεράκια «*» δηλώνουν τον κόμβο που κάθε φορά διαλέξαμε. Έτσι στο συγκεκριμένο γράφο αρχίζοντας από το 1 ο δείκτης οδηγεί στο 2. Από το 2 πηγαίνουμε στο 4 και από εκεί αναγκαστικά οδηγούμαστε στο 3 αφού είναι το μονό σημείο που δεν έχουμε επισκεφθεί και τελικά ο κύκλος κλείνει στο 1 από όπου τελικά ξεκινήσαμε. Τελικά ο ζητούμενος κύκλος είναι 1→2→4→3 ,όπως φαίνεται στο παρακάτω σχήμα, και φυσικά δεν παίζει ρόλο ο κόμβος από τον οποίο θα ξεκινήσουμε.

3.4 Υπολογισμός n-οστού αριθμού της ακολουθίας Fibonacci

Η ακολουθία Φιμπονάτσι (Fibonacci) είναι μία ακολουθία αριθμών που ονομάζονται αριθμοί Φιμπονάτσι. Έλαβε το όνομά της από τον Λεονάρντο της Πίζας (προσωνύμιο Φιμπονάτσι). Οι όροι της ακολουθίας ορίζονται από τον εξής αναδρομικό τύπο:

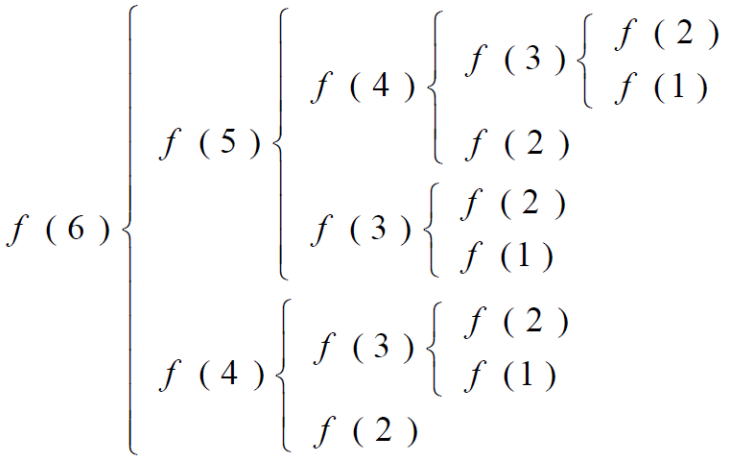
$F_n = F_{n-1} + F_{n-2}$,με $F_0=0$ και $F_1=1$

(Οι πρώτοι όροι της ακολουθίας είναι: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89)

Εδώ είναι μια απλή συνάρτηση (γραμμένη σε ψευδογλώσσα) που βρίσκει τον n-οστό αριθμό της ακολουθίας Fibonacci, βασισμένη άμεσα στον μαθηματικό καθορισμό.

```
function f(n)
    if n = 0 return 0
    if n = 1 return 1
    return f(n − 1) + f(n − 2)
```

Για να βρούμε τον n-οστό αριθμό της ακολουθίας Fibonacci (π.χ. τον 6^{ο} αριθμό), προχωρώντας «από το τέλος προς την αρχή», θα κάναμε τα εξής βήματα όπως φαίνονται στο παρακάτω σχήμα



Από το παραπάνω σχήμα βλέπουμε ότι το f(1) και το f(2) υπολογίστηκαν 3 και 4 φορές αντίστοιχα από την αρχή. Στα μεγαλύτερα παραδείγματα, οι πολλές περισσότερες τιμές f(), ή τα υποπροβλήματα, υπολογίζονται εκ νέου, οδηγώντας σε μεγαλύτερη καθυστέρηση εκτέλεσης πράξεων και κυρίως στην καθυστέρηση εύρεσης της λύσης.

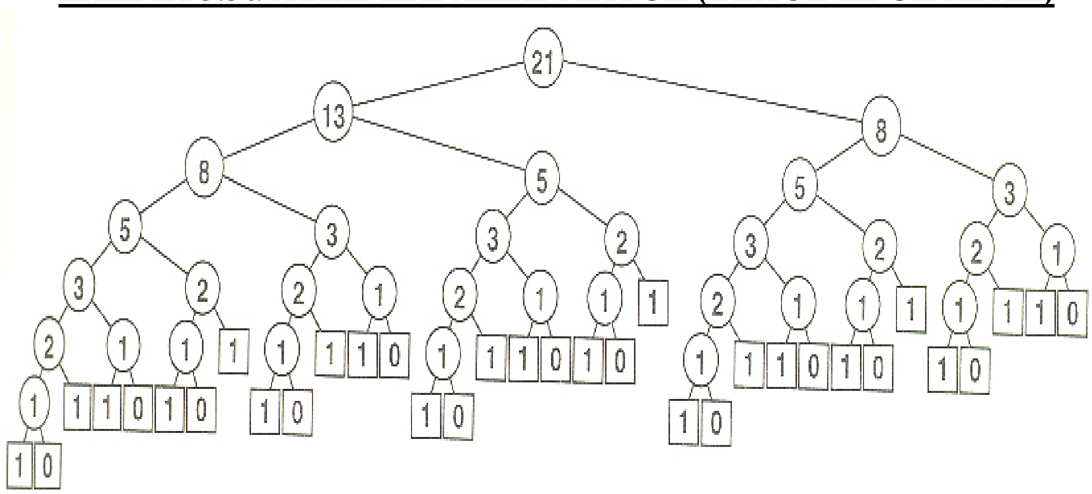
Τώρα, χρησιμοποιώντας έναν απλό χάρτη (map) ,τον m[], δηλαδή έναν πίνακα στον οποίο αποθηκεύεται σε κάθε στοιχειό του κάθε αποτέλεσμα της f() που έχει υπολογιστεί ήδη, και τροποποιώντας τη συνάρτηση που γράψαμε παραπάνω και αναβαθμίζοντας την, φτάνουμε στην παρακάτω συνάρτηση.

```
var m := map(0 → 0, 1 → 1)
  function f(n)
     if map m does not contain key n
        m[n] := f(n − 1) + f(n − 2)
     return m[n]
```

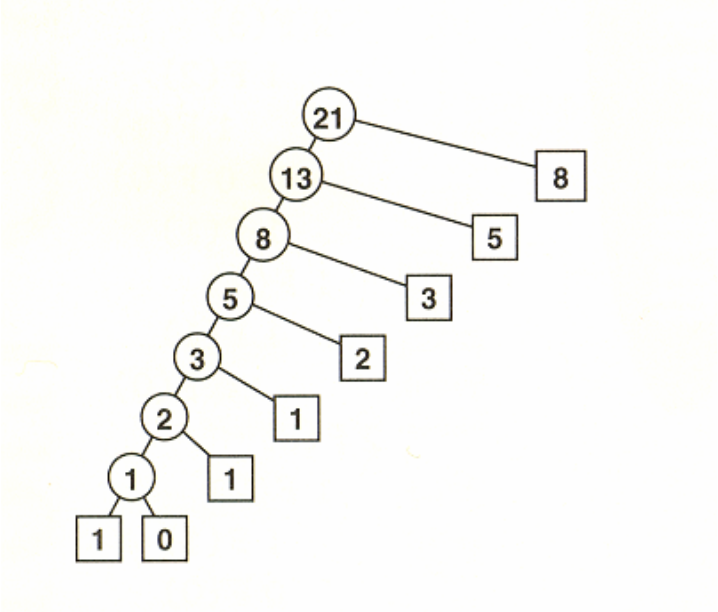
Αυτή η τεχνική ,το να αποθηκεύουμε ,δηλαδή , τις τιμές που έχουν υπολογιστεί ήδη καλείται **memorization** ,αυτό είναι ο αναδρομικός υπολογισμός, δεδομένου ότι σπάζουμε αρχικά το πρόβλημα στα υποπροβλήματα και έπειτα υπολογίζουμε και αποθηκεύουμε τις τιμές.

Από τα δυο επόμενα σχήματα βλέπουμε ακόμα καλύτερα την διαφορά του υπολογισμού ξεκινώντας από το τέλος προς την αρχή (Σχήμα 3.3α) και με χρήση της αναδρομικής σχέσης σε συνδυασμό με τον πίνακα χάρτη m[] (Σχήμα 3.3β). Επίσης παρατηρούμε ότι με την χρήση του δυναμικού προγραμματισμού υπολογίσαμε μόνο μια φορά το f(1) και το f(2) σε σχέση με την αρχική μέθοδο υπολογισμού(από το τέλος προς την αρχή) που τα υπολογίσαμε 3 και 4 φορές αντίστοιχα το καθένα.

ΣΧΗΜΑ 3.3α ΛΥΣΗ ΠΑΡΑΔΕΙΓΜΑΤΟΣ (ΤΕΛΟΣ ΠΡΟΣ ΑΡΧΗ)



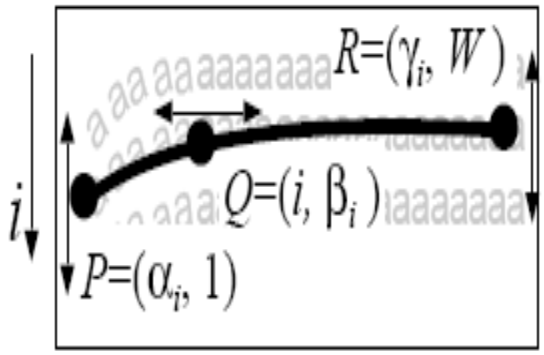
ΣΧΗΜΑ 3.3β ΛΥΣΗ ΠΑΡΑΔΕΙΓΜΑΤΟΣ (ΜΕ ΧΡΗΣΗ ΠΙΝΑΚΑ m[])



3.5 Τεχνική Ezaki

Το μοντέλο σκέβρωσης της εικόνα εκτιμάται με τη βοήθεια ενός συνόλου από cubic splines (καμπύλες γραμμές με τρία σημεία ελέγχου), όπως φαίνεται στο παρακάτω σχήμα (Σχήμα 3.4). Κάθε cubic spline ταιριάζετε μη γραμμικά με μια γραμμή κειμένου ή με ένα κενό ανάμεσα στις γραμμές. Τα splines γίνονται βέλτιστα καθολικώς (optimized globally), δηλαδή προσαρμόζονται ανάλογα για να βελτιστοποιούν την όλη εικόνα και όχι τη γειτονία στην οποία βρίσκονται. Αυτό έχει ως αποτέλεσμα τα splines να αλληλοεξαρτώνται για την βελτιστοποίησή τους.

ΣΧΗΜΑ 3.4 ΚΑΜΠΥΛΗ ΓΡΑΜΜΗ ΜΕ ΤΡΙΑ ΣΗΜΕΙΑ ΕΛΕΓΧΟΥ



Για κάθε εικόνα με ύψος H (Height) και πλάτος W (Width), χρησιμοποιούνται H το πλήθος cubic splines. Κάθε cubic spline ελέγχεται από τρία σημεία:

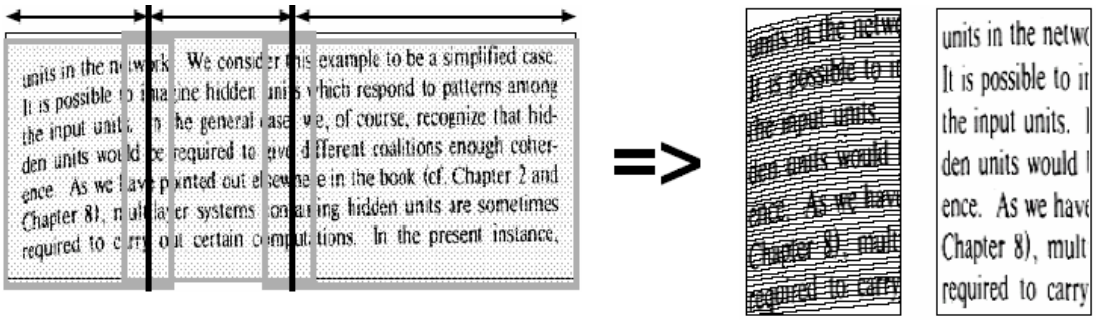
1. P(αi,1), κινείται κάθετα στην πρώτη στήλη κατά αi.
2. Q(i,βi), κινείται οριζόντια κατά βi στην γραμμή i.
3. R(γi,W), κινείται κάθετα στην τελευταία (W-ιοστή) στήλη.

Υπάρχουν βέβαια μερικοί περιορισμοί για τα μεγάλα κενά και τις τομές ανάμεσα σε διπλανά splines. Πρέπει να τονίσουμε ότι παράλληλα αυξάνεται η ευρωστία για τις διάφορες τοπικές ανωμαλίες που εμφανίζονται). Οι περιορισμοί είναι οι εξής:

$$0 \le \alpha_i - \alpha_{i-1} \le 2 \quad , \quad -1 \le \beta_i - \beta_{i-1} \le 1 \quad \text{και} \quad 0 \le \gamma_i - \gamma_{i-1} \le 2$$

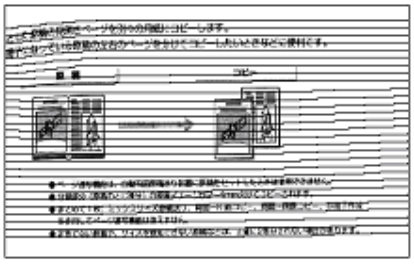
Για την καθολική βέλτιστη μοντελοποίηση της καμπυλότητας, χρησιμοποιείται δυναμικός προγραμματισμός. Διαχωρίζουμε την εικόνα σε υπο-εικόνες και εφαρμόζουμε την τεχνική Ezaki σε συνδυασμό με την μέθοδο του δυναμικού προγραμματισμού σε κάθε υπο-εικόνα για να έχουμε καλύτερα αποτελέσματα, όπως φαίνεται στο παρακάτω σχήμα.

ΣΧΗΜΑ 3.5 ΔΙΑΧΩΡΙΣΜΟΣ ΕΙΚΟΝΑΣ ΣΕ ΥΠΟ-ΕΙΚΟΝΕΣ

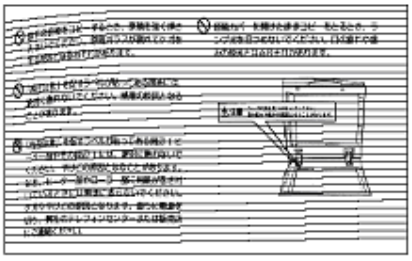


Στα παρακάτω σχήματα (Σχήμα 3.6) βλέπουμε εφαρμογή της τεχνικής Ezaki σε εικόνες που χρειάζονται μορφοποίηση. Στο πάνω μέρος είναι οι αρχικές εικόνες προς επεξεργασία και στο κάτω μέρος οι εικόνες μετά την μορφοποίηση.

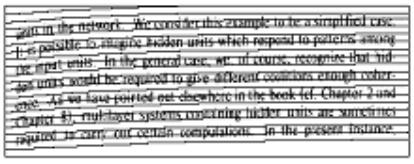
ΣΧΗΜΑ 3.6 ΕΦΑΡΜΟΓΗ ΤΗΣ ΤΕΧΝΙΚΗΣ EZAKI



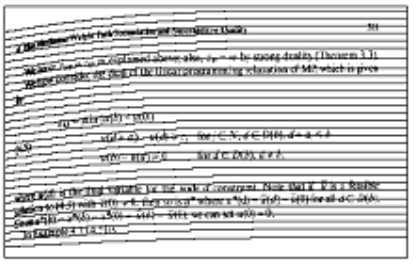
(α)



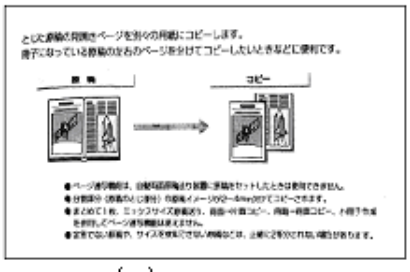
(β)



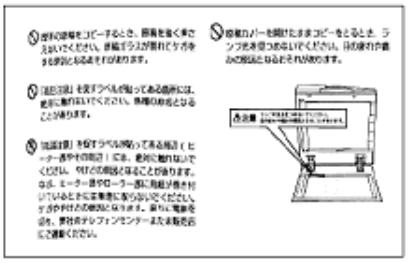
(γ)



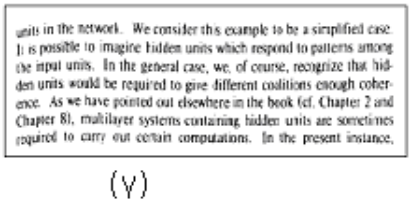
(δ)



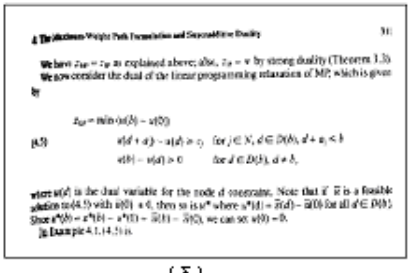
(α)



(β)



(γ)



(δ)

Κεφάλαιο 4ο
(Εφαρμογές / Applications)

4.1. Πρόβλημα σάκου

Σε αυτό το σημείο ας θυμηθούμε ότι το πρόβλημα του σάκου ορίζεται ως να γεμίσουμε ένα σάκο που έχει χωρητικότητα W. Από μία λίστα με n αντικείμενα πρέπει να επιλέξουμε τα αντικείμενα που πρόκειται να τοποθετηθούν στο σάκο. Κάθε αντικείμενο έχει ένα βάρος w_i και ένα κέρδος p_i . Σε μια εφικτή πλήρωση του σάκου το άθροισμα των βαρών των αντικειμένων που έχουν εισαχθεί δεν ξεπερνά τη χωρητικότητα του σάκου.

Στον παρόν κεφάλαιο θα παρουσιαστούν 2 υλοποιήσεις του προβληματος του σάκου:

Η πρώτη υλοποίηση είναι ένας απλοϊκός κώδικας με σκοπό την επεξήγηση και κατανόηση της λειτουργίας του προβλήματος του σάκου. Η δεύτερη υλοποίηση είναι ένας κώδικας πιο περίπλοκος από την MathWorks γραμμένος από τον Petter Strandmark, ο οποίος προσφέρει μεγαλύτερο βαθμό ελευθερίας επίλυσης του προβλήματος του σάκου

4.1.1 1η υλοποίηση

Χρησιμοποιώντας την θεωρία που αναλύσαμε στο 3ο κεφάλαιο για το πρόβλημα σάκου, το παράδειγμα που δώσαμε και με βάση την επαναληπτική λύση, δημιουργήσαμε τον παρακάτω κώδικα. Ο κώδικας είναι γραμμένος σε γλώσσα προγραμματισμού C++, και θα αναλυθεί παρακάτω κάθε κομμάτι του κώδικα.

Λεπτομέρειες παραδείγματος

```
#include <stdio.h>

int max(int a, int b){

        if(a>b)
                return a;
        else
```

```
        return b;

}

void main()
{
        int w[5];
        int u[5];
        int V[5][6];
        int i,j;

        w[0]=u[0]=0;

        printf("I xoritikotita tou sakou einai 5.\n\n");

        for(i=1;i<5;i++){
                printf("Dose to varos tou %dou antikeimenou:",i);
                scanf("%d",&w[i]);
                printf("\n");
        }


        for(i=1;i<5;i++){
                printf("Dose tin aksia tou %dou antikeimenou:",i);
                scanf("%d",&u[i]);
                printf("\n");
        }

        for(i=0; i<5; i++)
                for(j=0; j<6; j++)
                        V[i][j]=0;

        for(i=1; i<5; i++){
                for(j=0; j<6;j++){
                   if(w[i] <= j){
                        V[i][j]=max(V[i-1][j],u[i]+V[i-1][j-w[i]]);}
                   else{
                        V[i][j]=V[i-1][j];}
                        }

        }
```

```
printf("O pinakas pou prokiptei einai\n");
        printf("----------------------------\n");

        for(i=0; i<5; i++){
                for(j=0; j<6; j++){
                        printf("%5d",V[i][j]);
        }
                printf("\n");
        }

        printf("\n");

        printf("H veltisti lisi einai: %d\n\n",V[i-1][j-1]);


}
```

Χρειαζόμαστε μια συνάρτηση την max() που δέχεται σαν ορίσματα δυο ακέραιους αριθμούς τον a και τον b και σαν έξοδο έχει τον μεγαλύτερο τον δυο.

```
int max(int a, int b){
        if(a>b)
                return a;
        else
                return b;
}
```

Δηλώνουμε δυο μονοδιάστατους πίνακες τον w[5] και τον u[5] ,τους οποίους χρησιμοποιούμε για να αποθηκεύσουμε τα βάρη και την αξία ,αντίστοιχα ,του κάθε αντικειμένου. Και οι δυο πίνακες έχουν 5 στοιχεία γιατί έχουμε 4 αντικείμενα και το πρώτο στοιχείο και των δυο πινάκων είναι 0 ,για ευκολία του προγράμματος.

```
int w[5];
int u[5];
```

και

```
w[0]=u[0]=0;
```

Επίσης, δηλώνουμε ένα δυσδιάστατο πίνακα τον V[5][6] για να αποθηκεύουμε τις βέλτιστες τιμές ανά επανάληψη συμφώνα με την θεωρία. Έχει 5 γραμμές, όσα είναι δηλαδή τα αντικείμενα που έχουμε στην διάθεση μας συν το πρώτο στοιχείο ,που όπως και των άλλων δυο πινάκων που δηλώσαμε, είναι το μηδενικό. Έχει 6 στήλες, όση είναι η χωρητικότητα του σάκου, συν το πρώτο στοιχείο κάθε στήλης που είναι το μηδενικό, για την ευκολία του προγράμματος.

```
int V[5][6];
```

Χρησιμοποιούμε δυο βρόγχους for και οι δυο για τον ίδιο σκοπό. Έναν for για να δηλώσει ο χρήστης του προγράμματος το βάρος του κάθε στοιχείου και άλλη μια για να δηλώσει την αξία του κάθε στοιχείου.

```
for(i=1;i<5;i++){
        printf("Dose to varos tou %dou antikeimenou:",i);
        scanf("%d",&w[i]);
        printf("\n");
}
```

Και

```
printf("Dose tin aksia tou %dou antikeimenou:",i);
scanf("%d",&u[i]);
```

Με την χρήση ενούς διπλού βρόγχου for (εμφολευμενές for) μηδενίζουμε τα περιεχόμενα όλων των στοιχείων του πίνακα V[5][6], για να μπορέσουμε να αποθηκεύσουμε και να ελέγξουμε τα αποτελέσματα κάθε σταδίου.

```
for(i=0; i<5; i++)
  for(j=0; j<6; j++)
    V[i][j]=0;
```

Το παρακάτω κομμάτι είναι το πιο σημαντικό κομμάτι όλου του κώδικα. Λαμβάνοντας υπόψη μας την θεωρία που αναφέρθηκε στο παραπάνω κεφάλαιο για το πρόβλημα του σάκου ,και χρησιμοποιώντας ένα διπλό βρόγχο for ,κάνουμε τα περιεχόμενα του τρέχοντος στοιχείου (V[i][j]) του πίνακα V[][] ίσο με το μέγιστο ανάμεσα στα περιεχόμενα του V[i-1][j] και στο V[i-1][j-w[i]] συν το u[i] (φυσικά όπου w[i]=το βάρος του τρέχοντος αντικειμένου και όπου u[i]=η αξία του τρέχοντος αντικειμένου). Βέβαια όλα αυτά γίνονται μόνο όταν το βάρος του τρέχοντος αντικειμένου είναι μικρότερο από το διαθέσιμο βάρος

αποθήκευσης. Αλλιώς, αν δεν ισχύει ο παραπάνω περιορισμός κάνουμε τα περιεχόμενα του τρέχοντος στοιχείου ίσο με τα περιεχόμενα του προηγούμενου στοιχείου που έχει το ίδιο διαθέσιμο βάρος ,δηλαδή, το V[i-1][j].

```
for(i=1; i<5; i++){
        for(j=0; j<6;j++){
           if(w[i] <= j){
                V[i][j]=max(V[i-1][j],u[i]+V[i-1][j-w[i]]);}
           else{
                V[i][j]=V[i-1][j];}  }
```

Με την χρήση ενός διπλού βρόγχου for εμφανίζουμε τον τελικό πίνακα του παραδείγματος με όλα τα στοιχεία και τα περιεχόμενα τους αναλυτικά.

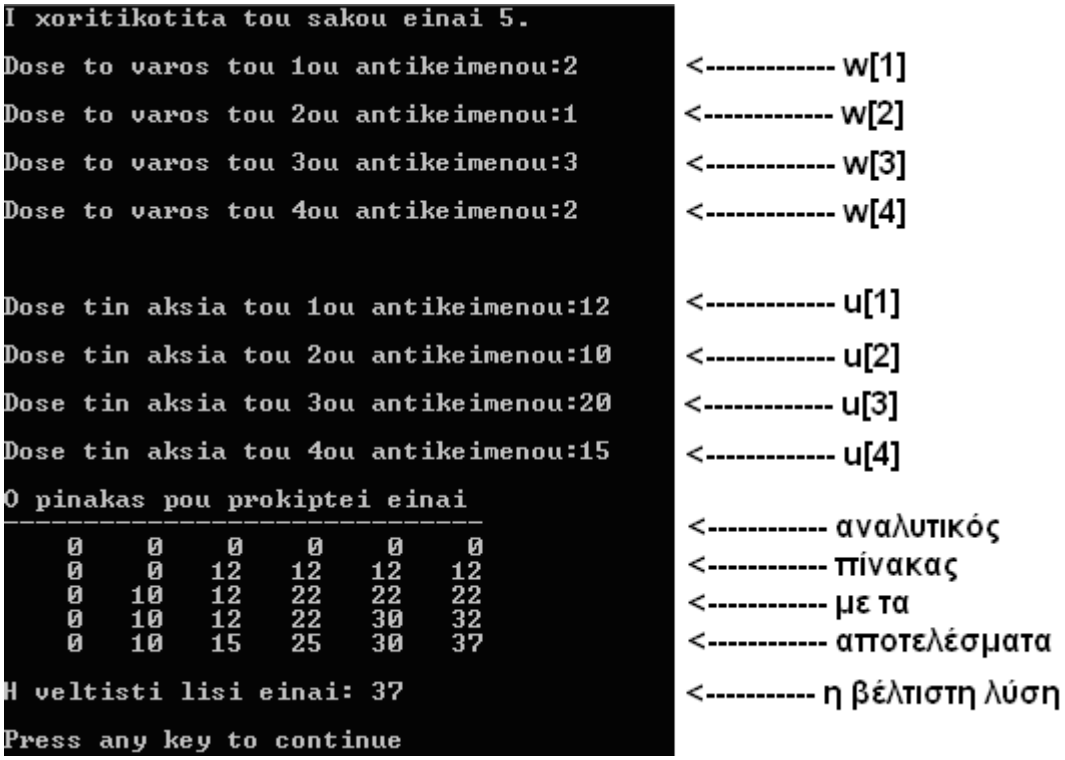
```
for(i=0; i<5; i++){
        for(j=0; j<6; j++){
                printf("%5d",V[i][j]);
        }
        printf("\n");
}
```

Τέλος, εμφανίζουμε την βέλτιστη λύση που βρίσκετε όπως αναφέρθηκε και στο προηγούμενο κεφάλαιο στα περιεχόμενα του τελευταίου στοιχείου του πίνακα ,δηλαδή στο V[5][6].

```
printf("H veltisti lisi einai: %d\n\n",V[i-1][j-1]);
```

Αν γίνει execute ο κώδικας θα παρατηρήσουμε την παρακάτω εικόνα

ΕΙΚΟΝΑ 4.1 ΕΙΚΟΝΑ ΕΞΟΔΟΥ 1ης ΥΛΟΠΟΙΗΣΗΣ



Αν συγκρίνουμε τα αποτελέσματα του προγράμματος με βάση τα δεδομένα μας με τον αναλυτικό πίνακα του παραδείγματος της θεωρίας του 3ου κεφαλαίου θα παρατηρήσουμε ότι είναι τα ίδια. Αυτό συμβαίνει γιατί όπως ειπώθηκε παραπάνω όλο το πρόγραμμα δημιουργήθηκε με βάση την επαναληπτική λύση και την αναδρομικότητα.

4.1.2 2η υλοποίηση

Η 2η υλοποίηση του αλγόριθμου του σάκου είναι ανεπτυγμένη με βάση την προηγούμενη εκδοχή ,με την χρήση του προγράμματος Matlab, αλλά με μερικές διαφορές

- Στην 2η υλοποίηση εμφανίζεται ένας πίνακας με 1 και 0 που αντιστοιχούν στα αντικείμενα που έχουν αποθηκευτεί ή όχι αντίστοιχα, ούτως ώστε να βελτιστοποιηθεί το κέρδος (σε αντίθεση με την 1η υλοποίηση που απλά αναφέρθηκε πως υπολογίζεται και ποιο είναι το κέρδος).
- Στην 2η υλοποίηση έχουμε δυναμικό μέγεθος πίνακα ανάλογα με τον αριθμό των τιμών που εισάγει ο χρήστης. Αντίθετα στην 1η υλοποίηση είναι δεδομένη η χωρητικότητα του πίνακα.

Για να εμφανιστεί ο πίνακας amount των 0 και 1 που χρησιμεύει στο να γνωρίζουμε ποια αντικείμενα αποθηκεύτηκαν , θα χρησιμοποιηθεί μια εμφωλεύμενη εντολή while μέσα σε μια while. Ο πίνακας αυτός είναι μονοδιάστατος με αριθμό στηλών ίσο με τον αριθμό των στοιχείων του πίνακα weights .Αρχικά γίνεται αρχικοποίηση του πίνακα, δηλαδή το περιεχόμενο όλων των στοιχείων του γίνεται 0.

```
amount = zeros(length(weights),1);
```

Πρέπει να δημιουργηθούν κάποιοι μετρητές και να τους αποδώσουμε τις τιμές ανάλογα με τον λόγω που δημιουργήθηκε ο καθένας. Ο μετρητής a=best (όπου best η βέλτιστη λύση) δημιουργήθηκε για να ελέγχεται αν με την χρήση του αντικειμένου που κάθε φορά διαλέγεται αυξάνουμε το κέρδος, με την αποθήκευση του, ή όχι. Ο μετρητής j= length(weights) (όπου length(weights) ο αριθμός των στοιχείων του πίνακα weights) χρησιμεύει σαν δείκτης του τελευταίου στοιχείου που θα χρησιμοποιηθεί ανά επανάληψη. Ο μετρητής Y=W (όπου W η χωρητικότητα του σάκου) χρησιμοποιείται για να τσεκάρουμε σε κάθε επανάληψη την τρέχουσα χωρητικότητα του σάκου σε περίπτωση που χρησιμοποιηθεί το τρέχον αντικείμενο.

```
a = best;
j = length(weights);
Y = W;
```

Επίσης ο πίνακας A() που υπάρχει σε αυτήν την υλοποίηση είναι ο πίνακας που αποθηκεύονται οι τιμές με την βέλτιστη λύση ανά επανάληψη. Όσο ο μετρητής **a** ,δηλαδή η βέλτιστη λύση ανά επανάληψη, και το τρέχον στοιχείο που ελέγχεται έχουν την ίδια τιμή τότε το τρέχον αντικείμενο για το οποίο γίνεται η εξωτερική επανάληψη πρέπει να αποθηκευτεί. Αλλιώς, ελέγχεται η τιμή του προηγούμενου στοιχείου (δηλαδή κινούμαστε στην ίδια στήλη για να συγκρίνουμε το τελευταίο κάθε φορά στοιχείο της γραμμής με τον μετρητή a).

```
while a > 0
   while A(j+1,Y+1) == a
      j = j - 1;
   end
   j = j + 1;
```

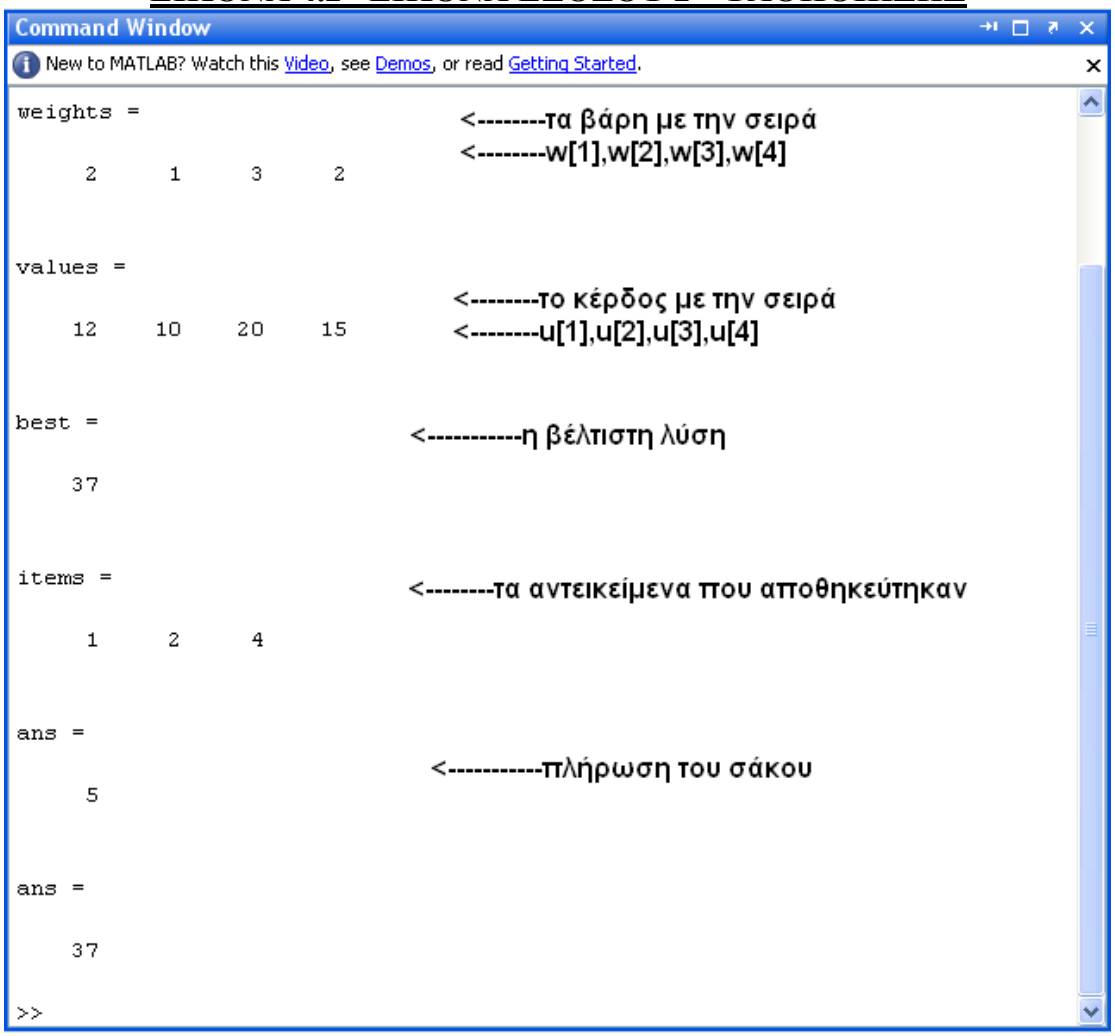
```
        amount(j) = 1;
        Y = Y - weights(j);
        j = j - 1;
        a = A(j+1,Y+1);
    end
```

Με βάση τα δεδομένα και της προηγούμενης υλοποίησης και για την εισαγωγή τους σε αυτήν την υλοποίηση τροποποιούνται ως εξής

```
weights = [2 1 3 2]
values = [12 10 20 15]
capacity = 5
```

Τα αποτελέσματα που προκύπτουν φαίνονται στην παρακάτω εικόνα

ΕΙΚΟΝΑ 4.2 ΕΙΚΟΝΑ ΕΞΟΔΟΥ 2ης ΥΛΟΠΟΙΗΣΗΣ



Αν συγκριθούν τα αποτελέσματα των δυο παραπάνω υλοποιήσεων παρατηρούμε ότι και στις δυο περιπτώσεις είναι όμοια τα αποτελέσματα. Στην $2^{η}$ περίπτωση βέβαια παρατηρούμε και ποια αντικείμενα αποθηκεύτηκαν ούτως ώστε να βελτιστοποιηθεί το κέρδος.

4.2 Εφαρμογή προβλήματος σάκου

Μια απλή εφαρμογή του αλγόριθμου του σάκου μπορεί να εφαρμοστεί και σε προβλήματα σε πολλούς κλάδους όπως για παράδειγμα στην αποδοχή request των χρηστών στο Rapidshare.

Έστω ότι σε μια δεδομένη χρονική στιγμή ο server του Rapidshare μπορεί να διαθέσει ένα δεδομένο bandwidth χωρητικότητας 200 αιτήσεων. Και δεδομένου ότι υπάρχουν 5 κατηγορίες χρηστών (Free, Premium1, Premium2, Premium3 και Premium4) ανάλογα με το αν θα καταθέσουν κάποια χρήματα και πόσα για να αγοράσουν κάποιες υπηρεσίες του server, κάθε κατηγορία χρηστών έχει ένα κέρδος όπως φαίνεται στον παρακάτω πίνακα.

		ΚΕΡΔΟΣ
ΧΡΗΣΤΕΣ	**Free**	**1**
	Premium1	**2**
	Premium2	**3**
	Premium3	**6**
	Premium4	**10**

Έστω ότι υπάρχουν την συγκεκριμένη χρονική στιγμή 10 χρήστες (3 της $1^{ης}$ κατηγορίας, 2 της $2^{ης}$, 1 της $3^{ης}$,2 της $4^{ης}$ και 2 της $5^{ης}$) και κάνουν αιτήσεις για χρήση των υπηρεσιών του server όπως φαίνεται στον παρακάτω πίνακα.

		ΑΙΤΗΣΕΙΣ
ΧΡΗΣΤΕΣ	**$1^{ος}$ Free**	**10**
	$2^{ος}$ Free	**13**
	$2^{ος}$ Free	**17**
	$1^{ος}$ Premium1	**25**
	$2^{ος}$ Premium1	**32**
	Premium2	**35**
	$1^{ος}$ Premium3	**12**
	$2^{ος}$ Premium3	**38**
	$1^{ος}$ Premium4	**10**

	2ος Premium4	**50**

Αν τροποποιηθούν κατάλληλα τα δεδομένα (δηλαδή στον πίνακα weights θα αποθηκευτούν οι τιμές των αιτήσεων και στον πίνακα values θα αποθηκευτούν το κέρδος από τον κάθε χρήστη ανάλογα με την κατηγορία που ανήκει)έτσι ώστε να γίνουν εισαγωγή στον αλγόριθμο του σάκου (της 2ης υλοποίησης) έχουμε τις εξής εισόδους

weights = [10 13 17 25 32 35 12 38 10 50]
και *values = [1 1 1 2 2 3 6 6 10 10]*

και να ορίσουμε, με βάση τα δεδομένα, ότι Capacity = 200 (διότι τόσες είναι οι αιτήσεις που μπορεί να απαντήσει ο server) , "τρέχοντας" τον αλγόριθμο έχουμε σαν έξοδο το εξής αποτέλεσμα:

ΕΙΚΟΝΑ 4.3 ΕΙΚΟΝΑ ΕΞΟΔΟΥ ΠΑΡΑΔΕΙΓΜΑΤΟΣ RAPIDSHARE

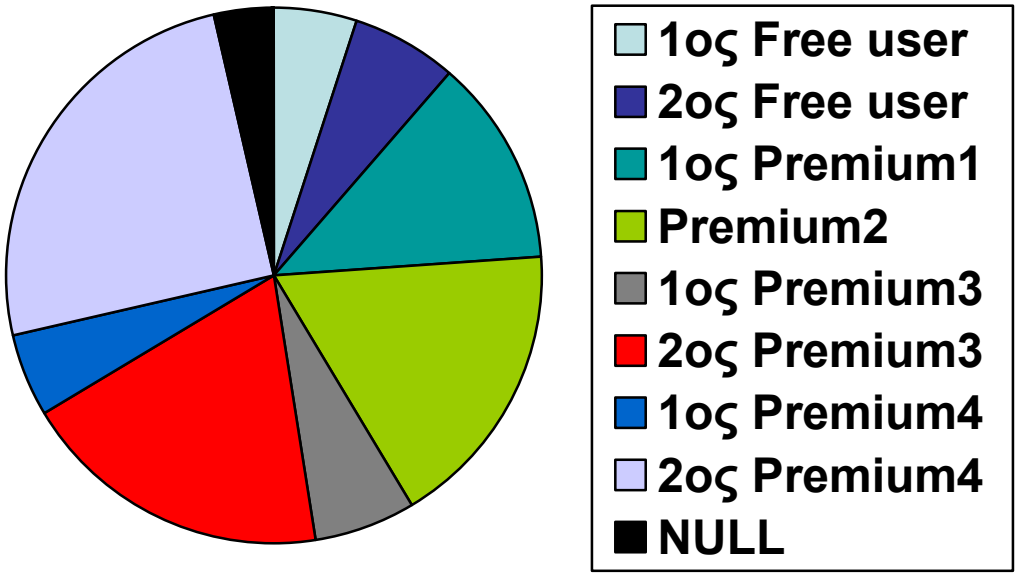
Εξετάζοντας καλύτερα την βέλτιστη λύση (δηλαδή το μέγιστο κέρδος με βάση τους χρήστες που έχουν εξυπηρετηθεί ανά κατηγορία) παρατηρούμε ότι εξυπηρετήθηκαν οι εξής χρήστες:

- Ο πρώτος Free user που έκανε 10 αιτήσεις και ο δεύτερος που έκανε 13.
- Ο πρώτος Premium1 user που έκανε 25 αιτήσεις
- Ο Premium2 user που έκανε 35 αιτήσεις
- Και οι δυο Premium3 users που έκαναν 12 και 38 αιτήσεις ο καθένας
- Και οι δυο Premium4 users που έκαναν 10 και 50 αιτήσεις ο καθένας

Το μέγιστο χρησιμοποιημένο bandwidth που προκύπτει είναι (10+13+25 +35+12+38+10+50=) 193.

Το διάγραμμα που προκύπτει από την κατανομή του bandwidth φαίνεται στην εικόνα

ΕΙΚΟΝΑ 4.4 ΔΙΑΓΡΑΜΜΑ ΚΑΤΑΝΟΜΗΣ BANDWIDTH



ΕΥΧΑΡΙΣΤΗΡΙΕΣ

Ευχαριστώ θερμά για το αμέριστο ενδιαφέρον του και τον πολύτιμο χρόνο που μου αφιερώθηκε από τον επιβλέποντα καθηγητή μου, κος Γεώργιος Τριανταφυλλίδης. Θέλω ,επίσης, να ευχαριστήσω την οικογένεια μου και τους φίλους συμφοιτητές μου για την συμπαράσταση και την βοήθεια τους.

ΒΙΒΛΙΟΓΡΑΦΙΑ & ΠΗΓΕΣ

- Dynamic Programming entry for consideration by the New Palgrave Dictionary of Economics John Rust, University of Maryland April 5, 2006
- Τεχνητή νοημοσύνη Βλαχάβας Ι. ,Κεφαλάς Π. , Βασιλειάδης Ν. ,Κόκκορας Φ. , Σακελλαρίου Η. ,Εκδόσεις Β. Γκιούρδας 2006
- Δυναμικός Προγραμματισμός Μηλιώτης Π. ,Εκδόσεις Οικονομικού Πανεπιστήμιου Αθηνών
- Διαφάνειες διαλέξεων του Καθηγητή D. P. Bertsekas του MIT http://www.mie.uth.gr/labs/pml/DP_Slides.pdf
- http://www.cs.uoi.gr/~stavros/Algorithms/DAA-2006-4.1.pdf.
- http://pelopas.uop.gr/~cstk02/dynamic.pdf
- Θεωρία του προβλήματος του σάκου http://en.wikipedia.org/wiki/Knapsack_problem
- Στοιχεία πιθανοθεωρίας Πανεπιστήμιο Πάτρας http://prlab.ceid.upatras.gr/courses/simeiwseis/DT/Stoixeia_Pithanothewrias.pdf
- Πρακτικά προβλήματα του ΔΠ στην σελίδα http://people.csail.mit.edu/bdean/6.046/dp/
- Ορισμός και λύση προβλήματος υπολογισμού ν-οστού αριθμού ακολουθίας Fibonacci http://www.technicalinterviewquestions.net/2009/01/fibonacci-sequence-dynamic-programming.html
- Σημειώσεις Σχεδίασης Αλγορίθμων Π. Κατσαρός Τμήμα πληροφορικής ΑΠΘ *delab.csd.auth.gr/~katsaros/Dynamic%20Programming.ppt*
- Dynamic Programming Wikipedia http://en.wikipedia.org/wiki/Dynamic_programming
- Αλγόριθμοι και πολυπλοκότητα Β. Ζησιμόπουλος Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθήνας Τμήμα πληροφορικής και τηλεπικοινωνιών 2008

ΕΥΡΕΤΗΡΙΟ ΣΧΗΜΑΤΩΝ ΚΑΙ ΕΙΚΟΝΩΝ